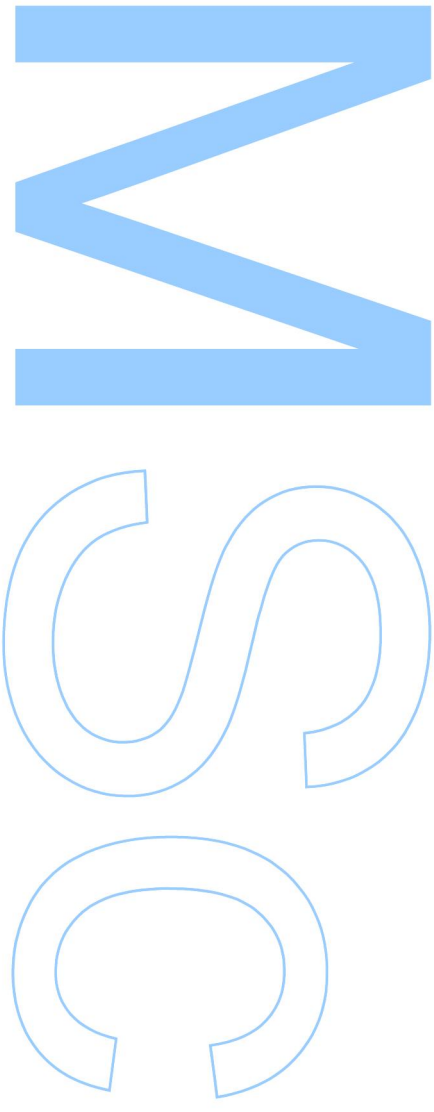
Static and Dynamic Types for Functional Languages

Pedro Jorge Fernandes Ângelo
Mestrado Integrado em Engenharia de Redes
e Sistemas Informáticos
Departamento de Ciência de Computadores
2017

Orientador
António Mário da Silva Marcos Florido, Professor Associado,
Faculdade de Ciências da Universidade do Porto

Coorientador
Pedro Baltazar Vasconcelos, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto



Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, / /

Abstract

The purpose of gradual typing is to integrate static and dynamic typing in a single program. Although many programming languages support either static or dynamic typing, these two approaches can be unified into a single mechanism to achieve gradual typing. In this dissertation we present gradual typing and the systems that enable it, such as gradual type systems and type inference algorithms for gradual typing, cast insertion rules for inserting runtime checks, as well as reduction rules that enable expression with runtime checks to be reduced. We implement many of these systems in Haskell, in particular a gradualizer that transforms static type systems into gradual ones and an interpreter for a gradually typed language. This dissertation also contains a new extension that allows the definition of gradual algebraic data types, which are standard algebraic data types with dynamic elements.

The first chapter covers many topics that aid in understanding the work done. These topics consist mainly of: a through study of many type systems, including the simply typed lambda calculus and Hindley-Milner type system, type inference algorithms and the difference between static and dynamic typing and the strengths and weaknesses of both.

Resumo

O propósito de tipagem gradual é integrar tipagem estática e dinâmica numa única linguagem. Embora muitas linguagens de programação suportam ou tipagem estática ou tipagem dinâmica, estas duas abordagens podem ser unificadas num único mecanismo para alcançar tipagem gradual. Nesta dissertação apresentamos tipagem gradual e os sistemas que a permitem, tal como sistemas de tipos graduais e algoritmos de inferência de tipos para tipagem gradual, regras de inserção de verificações para a inserção de testes de tipos durante tempo de execução, assim como regras de redução que permitem que expressões com verificações em tempo de execução sejam reduzidas. Nós implementamos muitos destes sistemas em Haskell, em particular um gradualizer que transforma sistemas de tipos estáticos em sistemas de tipos graduais e um interpretador para uma linguagem com tipagem gradual. Esta dissertação também contém uma nova extensão que permite a definição de tipos de dados algébricos graduais, que são tipos de dados algébricos normais com elementos dinâmicos.

O primeiro capítulo cobre vários tópicos que ajudam na compreensão do trabalho feito. Estes tópicos consistem sobretudo em: um estudo completo de muitos sistemas de tipos, incluindo cálculo lambda com tipagem simples e o sistema de tipos de Hindley-Milner, algoritmos de inferência de tipos e a diferença entre tipagem estática e dinâmica e as vantagens e desvantagens de ambas.

Contents

Chapter 1

Introduction

Programming languages adhere to one of two major typing disciplines: static typing or dynamic typing. Both disciplines have different strengths and weaknesses and thus, programming languages are often chosen for different tasks according to their typing disciplines. However, if these typing disciplines could be combined, their strengths could be harnessed while removing their weaknesses.

In statically typed languages, each term is assigned a type (either by the programmer or by means of a type inference algorithm) and that type is known during program compilation. Therefore, the type checker will ensure that all terms are given a correct type according to the rules of the type system, thus ensuring all type errors are caught before runtime. In dynamically typed languages, terms have no assigned type at compile time. Instead of checking for type errors in compilation time, these languages check for errors at run time. Statically typed languages have the advantage of being less prone to errors (due to type checking) and being more efficient (due to the lack of runtime checks) while dynamically typed languages have the advantage of allowing faster development (due to the lack of need from the programmer to consider the types of the terms).

Gradual typing combines these two typing disciplines. It allows the programmer to decide if certain parts of the program are to be statically or dynamically typed, thus allowing for the speed and efficiency of static typing and the adaptability and speed of development of dynamic typing in one program.

To illustrate this ability to choose between static and dynamic typing, consider the following example:

$$(\lambda x : \mathsf{Dyn} \;.\; 1 + x)\ \mathsf{true}$$

In order to switch between static and dynamic typing, a type annotation with type Dyn (dynamic) is inserted, as it is represented by the expression above. The function in the left takes an argument of unknown (dynamic) type and returns the addition between the value 1 and that argument. This way, we delay type checking between the type of the function and it's argument. What

happens is that the gradual type system will check if type Dyn is consistent with type Bool (Dyn $\sim$ Bool) when comparing both types. According to the consistency relation ($\sim$), all types are consistent with the Dyn (dynamic) type. As such, this expression type checks, and has final type Int. However, reducing the expression will ultimately yield $1 + \mathsf{true}$ resulting in a type error. This type error then must be caught during runtime, through the insertion of type checks in operations that happen in dynamic code, as it is common in dynamically typed languages [32].

The contributions of this dissertation are:

1. An in-depth study of gradual typing and its components such as gradual types, the relations between types (equality and consistency) and type system rules that allow for gradual typing.
2. Several implementations, in Haskell, such as an algorithm for generating gradual type systems from static type systems (based on [9]) and an interpreter for a gradually typed language with a type inference module. The implementations are available in the following GitHub repositories: the implementation for The Gradualizer algorithm is available at https://github.com/pedroangelo/gradualizer while the implementation for the interpreter is available at https://github.com/pedroangelo/interpreter.
3. Extensions to enable gradual data types, which are regular algebraic data types, such as those from Haskell, with the possibility for dynamic elements [2].

This dissertation is structured as following. Section 2 features a detailed explanation of background work necessary to the development of this dissertation and will focus on several type systems, including the Simply Typed Lambda Calculus and Hindley-Milner type system. Type inference and gradual type systems will also be approached. Section 3 will present a thorough explanation of a methodology to automatically generate gradual type systems, The Gradualizer [9], and type inference for gradual typing, based on [14]. Section 4 will focus on the operational semantics for gradual typing. In this chapter, the implementation of a gradual language will also be presented. Finally, in Section 5 we present an extension to the type system that enables the definition of gradual algebraic data types. The work presented in this section is an original work from the author.

Chapter 2

Background

In this chapter we provide an overview of some theoretical basis that support this work. In summary, we will be addressing many type systems, including gradual type systems and several extensions, including polymorphism.

2.1 Lambda Calculus

The lambda-calculus, shown in Figure 2.1, was first introduced in [6, 8]. Good surveys can be found in [5, 23]. It forms the core language of many programming languages, and it provides a system for the representation of computer programs that at the same time can be reasoned with in a mathematical sense [4]. It defines a set of basic operations that can be used to encode all computer programs. The lambda-calculus does not have numbers, arithmetic operations or even conditional statements, however it is easy to code these as lambda expressions or to add these features as extensions.

Syntax

$$\begin{array}{lrcl} \textit{Expressions } e & ::= & x \mid \lambda x\ .\ e \mid e\ e \\ \textit{Values} \qquad v & ::= & \lambda x\ .\ e \end{array}$$

Evaluation (β-reduction)

$$(\lambda x\ .\ e)\ v \longrightarrow [x \mapsto v]\ e$$

Figure 2.1: Lambda-Calculus (λ)

2.1.1 Syntax

The lambda-calculus is composed of 3 terms, as shown in Figure 2.1. A variable x; the abstraction of a variable x in a term e, written as $\lambda x\ .\ e$; and the application of a term e_1 to another term e_2, written as $e_1\ e_2$. An abstraction is a nameless function with only one argument x and whose body is e. An application is the application of a term with another term, and it can be used to simulate the application of a function (abstraction) with an argument (another term). Considering the example of a mathematical formula, the function can be written in the lambda-calculus[1] as

$$\lambda n_1\ .\ \lambda n_2\ .\ (n_1 * 43) + n_2 + 2$$

This expression is a function that accepts two arguments, since there are two abstractions, and returns the result of the formula according to the two arguments. Applying it to the arguments is written as:

$$((\lambda n_1\ .\ \lambda n_2\ .\ (n_1 * 43) + n_2 + 2)\ 2)\ 3$$

2.1.2 Reduction

Using the syntax described in Figure 2.1, a programmer can write programs. However, there is the need to run these programs and obtain the result of the computation. The lambda-calculus has reduction rules that reduce expressions. The main rule is β-reduction.

β-reduction reduces the application of an abstraction $\lambda x\ .\ e$ with some other term v by replacing the bounded variable x with term v in the body of the abstraction e.

Reducing the expression given as an example previously returns the value 91.

$$\begin{aligned}
&\underline{((\lambda n_1\ .\ \lambda n_2\ .\ (n_1 * 43) + n_2 + 2)\ 2)}\ 3 \longrightarrow && (\beta\text{-reduction})\\
&\underline{(\lambda n_2\ .\ (2 * 43) + n_2 + 2)\ 3} \longrightarrow && (\beta\text{-reduction})\\
&(2 * 43) + 3 + 2 \longrightarrow 91 && (\text{Arithmetic rules})
\end{aligned}$$

There are several important features and restrictions of β-reduction and the lambda-calculus itself. An extensive presentation of the subject can be found in [5].

2.2 Type Systems

A type system is a set of rules that assigns to every term a property called type. The main purpose of a type system is to check for type errors (terms that were combined in an incorrect manner according to their types), thus ensuring no errors will occur when the program runs.

[1] As stated above the lambda-calculus does not support integers or arithmetic operators, however, and for the sake of this example, we will consider an extension to the lambda-calculus that enables these primitives.

Syntax

$$\begin{array}{lrcl} \textit{Expressions} & e & ::= & x \mid \lambda x : T \;.\; e \mid e\ e \\ \textit{Values} & v & ::= & \lambda x : T \;.\; e \\ \textit{Types} & T & ::= & T \to T \\ \textit{Context} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \end{array}$$

$\boxed{\Gamma \vdash e : T}$ Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var} \qquad \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1 \;.\; e : T_1 \to T_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}\ \text{T-App}$$

$\boxed{e \longrightarrow e}$ Evaluation

$$\frac{}{(\lambda x : T \;.\; e)\ v \longrightarrow [x \mapsto v]e}\ \text{E-AppAbs}$$

Figure 2.2: Simply Typed Lambda Calculus ($\lambda_{\to}$)

2.2.1 Simply Typed Lambda Calculus

The most elementary of all type systems is the simply typed lambda calculus [7, 11], shown in Figure 2.2. This system was introduced in order to prevent paradoxical expressions in the lambda calculus [6, 8].

It is composed by three type rules, each responsible for typing one of the three different terms in the lambda calculus (variable, abstraction and application). Rule T-Var first searches the context Γ for the type of the variable x. Assuming that that type is T, it types the variable x with type T. Rule T-Abs first tries to type the term e, assuming that x has type T_1. Assuming e types with type T_2, it then types the abstraction of the variable x from a term e with type $T_1 \to T_2$. Rule T-App first tries to type the terms e_1 and e_2. Assuming that both terms type (that $e_1 : T_1 \to T_2$ and that $e_2 : T_1$), it then types the application of term e_1 with term e_2 with type T_2. Consider the following expression:

$$app \triangleq \lambda f : T_1 \to T_2 \;.\; \lambda x : T_1 \;.\; f\ x$$

The operator $\triangleq$ denotes an equality between two expressions, and is used throughout this dissertation to assign names to expressions. As the name indicates, this expression mimics the application of a function (f with type $T_1 \to T_2$) to an argument (x with type T_1). The following typing derivation illustrates the explanation above. The final type of the expression is then $(T_1 \to T_2) \to T_1 \to T_2$.

$$\dfrac{\dfrac{}{f : T_1 \to T_2, x : T_1 \vdash f : T_1 \to T_2}\ \text{T-Var} \quad \dfrac{}{f : T_1 \to T_2, x : T_1 \vdash x : T_1}\ \text{T-Var}}{\dfrac{f : T_1 \to T_2, x : T_1 \vdash f\ x : T_2}{\dfrac{f : T_1 \to T_2 \vdash \lambda x : T_1\ .\ f\ x : T_1 \to T_2}{\vdash \lambda f : T_1 \to T_2\ .\ \lambda x : T_1\ .\ f\ x : (T_1 \to T_2) \to T_1 \to T_2}\ \text{T-Abs}}\ \text{T-Abs}}\ \text{T-App}$$

This system is simple and powerful, however certain extensions are necessary before it can be used to model real world computer programs. Many extension have been proposed to the simply typed lambda calculus along with the corresponding typing rules, such as base types, like Bool and Int, and other constructs like references, let bindings and pairs [23].

2.2.2 The Hindley-Milner Type System

This type system allows polymorphism in the lambda calculus in a specific form: let-polymorphism. It was introduced in [18] by J. Roger Hindley and it was latter rediscovered by Robin Milner in [20]. Luis Damas also contributed by defining a type inference algorithm, Algorithm W [12, 13]. Figure 2.3 provides the syntax and typing rules.

The main difference between this and other conventional type systems (such as the STLC) is the separation of types into two different classes: monomorphic and polymorphic types. Monomorphic types, also known as monotypes, represent only a specific type such as a base type (e.g. Int or Bool). Type variables are also considered monotypes, behaving as base types with unknown identity. Polymorphic types, also known as polytypes or type schemes, are types where type variables are bound by for-all quantifiers. Unlike other systems, such as system F [23], for-all quantifiers only appear on the top level.

This system enables polymorphism in a specific manner, called let-polymorphism. The following expression $(\lambda f\ .\ (f\ \mathsf{true}, f\ 0))\ (\lambda x\ .\ x)$ cannot be typed, because f has a monomorphic type, and it cannot simultaneously be $\mathsf{Int} \to \mathsf{Int}$ and $\mathsf{Bool} \to \mathsf{Bool}$. However the expression $\mathsf{let}\ f = \lambda x\ .\ x\ \mathsf{in}\ (f\ \mathsf{true}, f\ 0)$ has type $(\mathsf{Bool}, \mathsf{Int})$. This is due to the fact that f is introduced through a let-expression, allowing it to be treated as a polymorphic variable. Then, according to the rule Inst, the type of $\lambda x\ .\ x$ can be instantiated to $\mathsf{Int} \to \mathsf{Int}$ or $\mathsf{Bool} \to \mathsf{Bool}$. $\sqsubseteq$ denotes the instance relation where a monomorphic type is a instance of a polymorphic type.

Consider the following expression:

$$\mathsf{let}\ i\ =\ \lambda x\ .\ x\ \mathsf{in}\ i\ i$$

In other type systems (such as the STLC) this expression cannot be typed, because self application cannot be typed. However, the Hindley-Milner type system allows this expression to be typed. What happens is that $\lambda x\ .\ x$ is typed with the polymorphic type $\forall\alpha.\alpha \to \alpha$. Then, when typing the application $i\ i$, the expression on the left is instantiated with type $(\alpha \to \alpha) \to \alpha \to \alpha$ while the expression on the right is instantiated with type $\alpha \to \alpha$. The following typing derivation

Syntax

$$\begin{array}{lrcl} \textit{Expressions} & e & ::= & x \mid \lambda x\ .\ e \mid e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \\ \textit{Monomorphic Types} & \tau & ::= & \alpha \mid \tau \to \tau \\ \textit{Polymorphic Types} & \sigma & ::= & \tau \mid \forall\alpha.\sigma \\ \textit{Context} & \Gamma & ::= & \emptyset \mid \Gamma, x : \sigma \end{array}$$

Specialization

$$\frac{\tau' = [\alpha_i \mapsto \tau_i]\tau \qquad \beta_i \notin free(\forall\alpha_1 \ldots \forall\alpha_i.\tau)}{\forall\alpha_1 \ldots \forall\alpha_n.\tau \sqsubseteq \forall\beta_1 \ldots \forall\beta_m.\tau'}$$

$\boxed{\Gamma \vdash e : \tau}$ Typing

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \textsc{Var} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x\ .\ e : \tau \to \tau'}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}\ \textsc{App}$$

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}\ \textsc{Let} \qquad \frac{\Gamma \vdash e : \sigma' \qquad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma}\ \textsc{Inst}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin free(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}\ \textsc{Gen}$$

Figure 2.3: Hindley-Milner (HM)

demonstrates this concept. The final type of the expression is then $\alpha \to \alpha$.

$$1\ \dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}\ \textsc{Var}}{\vdash (\lambda x.x) : \alpha \to \alpha}\ \textsc{Abs}}{\vdash (\lambda x.x) : \forall\alpha.\alpha \to \alpha}\ \textsc{Gen}$$

$$2\ \dfrac{\dfrac{\dfrac{}{i : \forall\alpha.\alpha \to \alpha \vdash i : \forall\alpha.\alpha \to \alpha}\ \textsc{Var}}{i : \forall\alpha.\alpha \to \alpha \vdash i : (\alpha \to \alpha) \to (\alpha \to \alpha)}\ \textsc{Inst} \qquad \dfrac{\dfrac{}{i : \forall\alpha.\alpha \to \alpha \vdash i : \forall\alpha.\alpha \to \alpha}\ \textsc{Var}}{i : \forall\alpha.\alpha \to \alpha \vdash i : \alpha \to \alpha}\ \textsc{Inst}}{i : \forall\alpha.\alpha \to \alpha \vdash i\ i : \alpha \to \alpha}\ \textsc{App}$$

$$\dfrac{1\ \dfrac{}{\vdash (\lambda x.x) : \forall\alpha.\alpha \to \alpha}\ \textsc{Gen} \qquad 2\ \dfrac{}{i : \forall\alpha.\alpha \to \alpha \vdash i\ i : \alpha \to \alpha}\ \textsc{App}}{\vdash \mathsf{let}\ i\ =\ (\lambda x.x)\ \mathsf{in}\ i\ i : \alpha \to \alpha}\ \textsc{Let}$$

$\boxed{\Gamma \vdash e : T \mid_{\chi} C}$ Constraint Generation

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\emptyset} \emptyset}\ \text{C-Var} \qquad \frac{\Gamma, x : X \vdash e : T \mid_{\chi} C}{\Gamma \vdash \lambda x\ .\ e : X \to T \mid_{\chi \cup \{X\}} C}\ \text{C-Abs}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash e_2 : T_2 \mid_{\chi_2} C_2}{\Gamma \vdash e_1\ e_2 : X \mid_{\chi_1 \cup \chi_2 \cup \{X\}} C_1 \cup C_2 \cup \{T_1 \doteq T_2 \to X\}}\ \text{C-App}$$

Figure 2.4: Constraint Generation

Along with this ability to type expressions with polymorphic types, one of the most important properties is the decidability of type inference and the derivation of the most general types [13].

2.3 Type Inference

A type system is used to type check programs when each expression is typed with its type. A type inference algorithm deduces the type of expressions, without requiring those expressions to be annotated with their types. If an expression is ill-typed, the type inference algorithm will reject it and assign no type.

A type inference algorithm (following the approach from [34]) is composed of two parts: constraint generation and constraint unification (or solving). The constraint generation algorithm takes the expressions and produces a set of constraints that the constraint solving algorithm will then solve.

2.3.1 Constraint Generation

The first part of type inference is to generate constraints for typeability, which will be solved during constraint solving. These constraints will be generated for a given expression according to a constraint typing judgement. Figure 2.4 presents the constraint typing judgement $\Gamma \vdash e : T \mid_{\chi} C$, meaning that a given context Γ and expression e have type T under the set of constraints C, while χ contains the extra variables used to express the constraints. $\doteq$ denotes the equality constraint. When read from bottom to top, the constraint typing rules provide a procedure that, given a context Γ and expression e, calculates the type T and set of constraints C, which will then need to be solved.

$\boxed{C \upsilon S}$ Unification

$$\frac{}{\emptyset \upsilon S} \qquad \frac{C \upsilon S}{C \cup \{T \doteq T\} \upsilon S} \qquad \frac{C \cup \{T_{11} \doteq T_{21}, T_{12} \doteq T_{22}\} \upsilon S}{C \cup \{T_{12} \to T_{12} \doteq T_{21} \to T_{22}\} \upsilon S}$$

$$\frac{C \cup \{X \doteq T\} \upsilon S}{C \cup \{T \doteq X\} \upsilon S} \qquad \frac{[X \mapsto T](C) \upsilon S \qquad X \notin Vars(T)}{C \cup \{X \doteq T\} \upsilon S \circ [X \mapsto T]}$$

Figure 2.5: Unification

To demonstrate the generation of constraints consider the following expression:

$$(\lambda x \,.\, x)\ 1$$

To type this expression, an extra constraint generation rule is required to infer the type of integers:

$$\frac{}{\Gamma \vdash n : \mathsf{Int} \mid_{\emptyset} \emptyset}\ \text{C-Int}$$

$$\dfrac{\dfrac{\dfrac{}{x : X_1 \vdash x : X_1 \mid_{\emptyset} \emptyset}\ \text{C-Var}}{\vdash \lambda x \,.\, x : X_1 \to X_1 \mid_{\{X_1\}} \emptyset}\ \text{C-Abs} \qquad \dfrac{}{\vdash 1 : \mathsf{Int} \mid_{\emptyset} \emptyset}\ \text{C-Int}}{\vdash (\lambda x \,.\, x)\ 1 : X_2 \mid_{\{X_1\} \cup \{X_2\}} \{X_1 \to X_1 \doteq \mathsf{Int} \to X_2\}}\ \text{C-App}$$

The constraint generation types the expression with type X_2 and produces the constraints $\{X_1 \to X_1 \doteq \mathsf{Int} \to X_2\}$.

2.3.2 Constraint Solving

The second part of type inference is to solve constraints and produce substitutions. Figure 2.5 presents the unification algorithm $C \upsilon S$, where a set of constraints C will be unified (υ) resulting in a set of substitutions S. For the Simply Typed Lambda Calculus and the Hindley-Milner type system, constraint solving is done by first order unification [24, 25]. X represents type variables and T represents types. $Vars(T)$ represents the set of all type variables in T. If the expression is typeable, a set of substitutions will be generated. These substitutions are then applied to the type provided by the constraint generation in order to obtain the final type of the expression. If the expression is not typeable, then the unification algorithm will not be able to unify the constraints, thus rejecting the expression.

Going back to the previous example, we may now attempt to unify the constraints $\{X_1 \to$

$\boxed{A \vdash_W e : \tau \mid S}$

$$\frac{x : \forall \alpha_1, ..., \alpha_n . \tau \in A}{A \vdash_W x : [\alpha_i \mapsto \beta_i]\tau \mid Id}\ \textsc{Var} \qquad \frac{A, x : \beta \vdash_W e : \tau \mid S}{A \vdash_W \lambda x\ .\ e : S\beta \to \tau \mid S}\ \textsc{Abs}$$

$$\frac{A \vdash_W e_1 : \tau_1 \mid S_1 \qquad S_1 A \vdash_W e_2 : \tau_2 \mid S_2 \qquad U(S_2\tau_1, \tau_2 \to \beta) = V}{A \vdash_W e_1\ e_2 : V\beta \mid VS_2S_1}\ \textsc{App}$$

$$\frac{A \vdash_W e_1 : \tau_1 \mid S_1 \qquad S_1 A, x : \overline{S_1 A}(\tau_1) \vdash_W e_2 : \tau_2 \mid S_2}{A \vdash_W \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \mid S_2 S_1}\ \textsc{Let}$$

Figure 2.6: Algorithm W

$X_1 \doteq \mathsf{Int} \to X_2\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset\ \upsilon\ \emptyset}}{\{X_2 \doteq \mathsf{Int}\}\ \upsilon\ [X_2 \mapsto \mathsf{Int}]}}{\{X_2 \doteq \mathsf{Int}, X_2 \doteq X_2\}\ \upsilon\ [X_2 \mapsto \mathsf{Int}]}}{\{X_1 \doteq \mathsf{Int}, X_1 \doteq X_2\}\ \upsilon\ [X_2 \mapsto \mathsf{Int}] \circ [X_1 \mapsto X_2]}}{\{X_1 \to X_1 \doteq \mathsf{Int} \to X_2\}\ \upsilon\ [X_2 \mapsto \mathsf{Int}] \circ [X_1 \mapsto X_2]}$$

Now that the constraints have been solved (or unified) we obtain a set of substitutions. Applying these substitutions to the type inferred by the constraint generations yields:

$$([X_2 \mapsto \mathsf{Int}] \circ [X_1 \mapsto X_2])\ X_2$$
$$[X_2 \mapsto \mathsf{Int}]\ X_2$$
$$\mathsf{Int}$$

Thus the expression has type Int.

2.3.3 Algorithm W

Algorithm W [13] is the type inference algorithm for the Hindley-Milner type system [18, 20]. Algorithm W is presented in Figure 2.6 and it uses the same syntax as that described in Figure 2.3, with $A \vdash e : \tau \mid S$ meaning that given a context A and an expression e, the algorithm will produce the type τ and the substitutions S. It uses the unification algorithm U from [25] and $\overline{A}(\tau) = \forall \alpha_1, ..., \alpha_n . \tau$, where $\alpha_1, ..., \alpha_n$ are variables occurring free in τ but not in A, and β represents fresh type variables. It is similar to the type inference algorithm for the STLC, despite it's original presentation style. A few differences are that, in algorithm W, constraint generation and constraint solving are combined in a single phase, whereas in the STLC's type inference algorithm, these two phases are completely separate.

Syntax

$$
\begin{array}{lrcl}
\textit{Expressions} & e & ::= & x \mid \lambda x\ .\ e \mid e\ e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\
\textit{Monomorphic Types} & \tau & ::= & \alpha \mid \tau \to \tau \\
\textit{Polymorphic Types} & \sigma & ::= & \tau \mid \forall \alpha.\sigma \\
\textit{Context} & \Gamma & ::= & \emptyset \mid \Gamma, x : \{\sigma, C\} \\
\textit{Constraints} & C & ::= & \emptyset \mid C \cup \{\tau \doteq \tau\}
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau \mid_{\chi} C}$ Constraint Generation

$$\frac{x : \{\forall \alpha_1, ..., \alpha_n.\tau', C'\} \in \Gamma \qquad \textit{instantiate}(\forall \alpha_1, ..., \alpha_n.\tau', C', \chi) = (\tau, C)}{\Gamma \vdash x : \tau \mid_{\chi} C}\ \text{C-Var}$$

$$\frac{\Gamma, x : \{\forall \emptyset.\alpha, \emptyset\} \vdash e : \tau \mid_{\chi} C}{\Gamma \vdash \lambda x\ .\ e : \alpha \to \tau \mid_{\chi \cup \{\alpha\}} C}\ \text{C-Abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash e_2 : \tau_2 \mid_{\chi_2} C_2}{\Gamma \vdash e_1\ e_2 : \alpha \mid_{\chi_1 \cup \chi_2 \cup \{\alpha\}} C_1 \cup C_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}}\ \text{C-App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \mid_{\chi_1} C_1 \qquad \textit{generalize}(\tau_1, \Gamma) = \sigma \qquad \Gamma, x : \{\sigma, C_1\} \vdash e_2 : \tau_2 \mid_{\chi_2} C_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \mid_{\chi_1 \cup \chi_2} C_1 \cup C_2}\ \text{C-Let}$$

Figure 2.7: HM(X)

Algorithm W combines constraint generation and constraint solving to form an algorithm that returns the most general type of the expression. However, the type inference style with two distinct phases might be more desirable. Algorithm W might be described with constraint generation and constraint solving as separate phases, however this causes a problem.

Types that are generalized may have type variables that are already constrained to a certain type. Therefore, when the type is instantiated, these constraints are lost because a new variable replaces the constrained variables. To prevent this problem, the constraints must be included in the context, and when instantiating type variables in the type, those same variables in the context must also be instantiated to the same fresh type variables. This solution is approached in HM(X) [21], shown in Figure 2.7. To solve constraints, the constraint unification algorithm from Figure 2.5 can be used. $\textit{instantiate}(\sigma, C', \chi) = (\tau, C)$ is the instantiation of the polymorphic type σ by replacing bound variables in σ with fresh variable and removing for all ($\forall$) quantifiers, resulting in the instance type τ. The type variables replaced in σ must also be replaced in C' by the same type variables as in σ, which results in C. χ denotes the new fresh variables used. $\textit{generalize}(\tau, \Gamma) = \sigma$ is the generalization of the type variables in τ that do not appear in Γ.

This results in the polymorphic type σ.

2.4 Gradual Type Systems

In gradual type systems, some terms have an unknown (or dynamic) type, meaning that some type errors cannot be caught during compilation, because the terms have not been assigned a type yet. Considering this, the gradual type system checks for type errors in the static parts of the program (those terms with a known, not dynamic, type) during compilation, and delays the remaining checks until runtime, through the insertion of runtime checks in operations that happen between non-static parts of the program.

Many attempts at gradual typing have been made in the past years [1, 3, 15–17, 22, 29–31] however, we will only focus on the Gradually Typed Lambda Calculus [9, 10, 26–28].

2.4.1 Gradually Typed Lambda Calculus

The gradually typed lambda calculus, first proposed by [26, 27], was developed with the purpose of achieving gradual typing. The key aspects of this system is the Dyn (dynamic) type, that represents the unknown type, and $\sim$ (consistency) relation, which checks whether two types are equal in the parts where both types are defined (defined as in known). Figure 2.8 depicts the GTLC.

In gradual typing, programs can be dynamically or statically typed. In order to change between dynamic and static typing, the use of the Dyn (dynamic) type is paramount. When an expression is dynamically typed, it must be annotated with the Dyn (dynamic) type. Otherwise, the expression must be annotated with a static type. Consider the example presented in Section 1:

$$(\lambda x : \mathsf{Dyn} \;.\; 1 + x)\ \mathsf{true}$$

The expression $(\lambda x : \mathsf{Dyn} \;.\; 1 + x)$ is dynamically typed while the expression true is statically typed. If the whole expression were statically typed, then according to the type rules of the STLC, it would contain a type error. The expression $(\lambda x \;.\; 1 + x)$ would be typed with type $\mathsf{Int} \to \mathsf{Int}$ and the expression true would be typed with type Bool, which goes against the typing rules of the type system, and is therefore a type error. However, using a gradual type system, the expression is typed with type Int. The typing derivation is the following:

An extra typing rule [14] is necessary for this expression:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad T_1 \sim \mathsf{Int} \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 \sim \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}\ \textsc{Add}$$

Syntax

$$
\begin{array}{llll}
\textit{Expressions} & e & ::= & x \mid c \mid \lambda x : T \,.\, e \mid e\ e \\
\textit{Constants} & c & ::= & n \mid \mathsf{true} \mid \mathsf{false} \\
\textit{Ground Types} & \gamma & ::= & \mathsf{Bool} \mid \mathsf{Int} \\
\textit{Types} & T & ::= & \gamma \mid \mathsf{Dyn} \mid T \to T \\
\textit{Context} & \Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}
$$

$\boxed{T \sim T}$ Consistency

$$\frac{}{\gamma \sim \gamma} \qquad \frac{}{\gamma \sim \mathsf{Dyn}} \qquad \frac{}{\mathsf{Dyn} \sim \gamma} \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$$

$\boxed{\Gamma \vdash e : T}$ Typing

$$\frac{}{\vdash n : \mathsf{Int}}\ \textsc{Int} \qquad \frac{}{\vdash \mathsf{true} : \mathsf{Bool}}\ \textsc{True} \qquad \frac{}{\vdash \mathsf{false} : \mathsf{Bool}}\ \textsc{False} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{Var}$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash \lambda x : T_1 \,.\, e : T_1 \to T_2}\ \textsc{Abs} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Dyn} \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : \mathsf{Dyn}}\ \textsc{App1}$$

$$\frac{\Gamma \vdash e_1 : T_1 \to T_3 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 \sim T_2}{\Gamma \vdash e_1\ e_2 : T_3}\ \textsc{App2}$$

Figure 2.8: Gradually Typed Lambda Calculus ($\lambda^{?}{}_{\to}$)

$$1\ \frac{\dfrac{\dfrac{}{\Gamma, x : \mathsf{Dyn} \vdash 1 : \mathsf{Int}}\ \textsc{Int} \qquad \mathsf{Int} \sim \mathsf{Int} \qquad \dfrac{}{\Gamma, x : \mathsf{Dyn} \vdash x : \mathsf{Dyn}}\ \textsc{Var} \qquad \mathsf{Dyn} \sim \mathsf{Int}}{\Gamma, x : \mathsf{Dyn} \vdash 1 + x : \mathsf{Int}}\ \textsc{Add}}{\vdash \lambda x : \mathsf{Dyn} \,.\, 1 + x : \mathsf{Dyn} \to \mathsf{Int}}\ \textsc{Abs}$$

$$\frac{1\ \dfrac{}{\vdash \lambda x : \mathsf{Dyn} \,.\, 1 + x : \mathsf{Dyn} \to \mathsf{Int}}\ \textsc{Abs} \qquad \dfrac{}{\vdash \mathsf{true} : \mathsf{Bool}}\ \textsc{True} \qquad \mathsf{Dyn} \sim \mathsf{Bool}}{\vdash (\lambda x : \mathsf{Dyn} \,.\, 1 + x)\ \mathsf{true} : \mathsf{Int}}\ \textsc{App2}$$

Although the expression was typed with final type Int, it obviously contains a type error. Consistency was used to compare types and it allowed the expression to be typed statically, despite the type error being present. However, the consistency relation also inserts runtime type checks to further check types at runtime. This way, although the expression was typed statically and the type error was not caught, during runtime the type error will be uncovered.

2.5 GHC Dynamic Types

Another related subject to our work was GHC's dynamic type [3] (included in Data.Dynamic package). This system provides an interface that allows programs to contain expressions of a dynamic type, featuring operations for conversion to and from the dynamic type.

```
toDyn :: Typeable a => a -> Dynamic
fromDyn :: Typeable a => Dynamic -> a -> a
fromDynamic :: Typeable a => Dynamic -> Maybe a
```

These conversion operations can be used to convert expressions to and from the dynamic type: *toDyn* converts the expression to the dynamic type, while *fromDyn* and *fromDynamic* attempt to convert the expression from the dynamic type. The expression (written in Haskell and using Data.Dynamic):

$$(\lambda x \ \rightarrow (fromDyn\ (toDyn\ x)\ 1) + 1)\ True \tag{2.1}$$

attempts to simulate the previous expression

$$(\lambda x : \mathsf{Dyn}\ .\ x + 1)\ \mathsf{true} \tag{2.2}$$

However, there are differences between the two expressions. The conversions from the dynamic type in Data.Dynamic must be accompanied with a default expression (which in the case of example 2.1 is the integer 1). This expression will replace the initial expression (true), in the conversion from dynamic, if the initial expression does not have the expected type. In the example above, the expression will then reduce to the integer 2. In the case of gradual typing, the expression 2.2 would instead generate a blame error. Furthermore, the expressions converted to dynamic in Data.Dynamic must have a type that belongs to the Typeable type class. For example, the expression

$$toDyn\ (\lambda x \rightarrow x)$$

cannot be expressed because $\lambda x \rightarrow x$ (with type $t \rightarrow t$) does not have a type that belongs to the Typeable type class. However, using gradual typing, the expression

$$(\lambda x : \mathsf{Dyn}\ .\ x)\ (\lambda x\ .\ x)$$

which is the analogous expression, has type Dyn.

Chapter 3

Gradual Types

3.1 Gradualizer

Gradual typing can be achieved by converting a static type system into a gradual type system by transforming the type system rules. This conversion is called *gradualization*. In previous approaches [26, 27], the introduction of the $\sim$ (consistency) relation and the Dyn (dynamic) type was crucial at constructing a gradual type system. However, *gradualizing* type systems is difficult and there was not a unified approach on how to do it. Different researchers may *gradualize* the same type system in different manners. Furthermore, how can researchers be sure they created a correct gradual type system?

To answer these questions, the Gradualizer [9] was proposed. The Gradualizer is a methodology for deriving gradual type systems from static type systems. It provides a series of steps, which analyse and transform a given static type system, leading to the deriving of a gradual type system. These steps also produce cast insertion rules that are used to insert casts in the language. These casts are used during runtime to check types.

Figure 3.1 shows the Gradually Typed Lambda Calculus, proposed in [9], which was obtained from the Simply Typed Lambda Calculus after following the steps of the methodology.

3.1.1 Compilation to cast calculus

The main purpose of The Gradualizer is to derive a gradual type system and a cast calculus compiler from a static type system. Cast calculus is the system that makes explicit the implicit casts introduced by consistency and pattern matching in the gradual type system. The cast calculus extends the STLC with the Dyn (dynamic) type and with a cast expression of the form $e : T_1 \Rightarrow^l T_2$, where T_1 represents the static type, l represents a blame label (which lets us know where the dynamic type error is) and T_2 represents the final type.

Syntax

$$
\begin{array}{lrcl}
\textit{Expressions} & e & ::= & x \mid \lambda x : T \,.\, e \mid e\ e \\
\textit{Types} & T & ::= & B \mid \mathsf{Dyn} \mid T \to T \\
\textit{Base Types} & B & ::= & \mathsf{Bool} \mid \mathsf{Int} \\
\textit{Context} & \Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}
$$

$\boxed{\Gamma \vdash_G e : T}$ Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash_G x : T}\ \text{T-Var} \qquad \frac{\Gamma, x : T_1 \vdash_G \ e : T_2}{\Gamma \vdash_G \lambda x : T_1 \,.\, e : T_1 \to T_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash_G e_1 : PM_1 \qquad PM_1 \rhd T_1 \to T_2 \qquad \Gamma \vdash_G e_2 : T_1' \qquad T_1' \sim T_1}{\Gamma \vdash_G e_1\ e_2 : T_2}\ \text{T-App}$$

$\boxed{T \sim T}$ Consistency

$$\frac{}{B \sim B} \qquad \frac{}{T \sim \mathsf{Dyn}} \qquad \frac{}{\mathsf{Dyn} \sim T} \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$$

$\boxed{T \rhd T}$ Pattern Matching

$$\frac{}{(T_1 \to T_2) \rhd T_1 \to T_2} \qquad \frac{}{\mathsf{Dyn} \rhd \mathsf{Dyn} \to \mathsf{Dyn}}$$

Figure 3.1: Gradually Typed Lambda Calculus ($\lambda^{?}{}_{\to}$)

The compilation to the cast calculus (also known as cast insertion) is responsible for inserting appropriate runtime casts and checks in points where the gradual type system used consistency or pattern matching to compare types. As the only types compared were static types, a runtime check is still necessary to deal with gradual types (those who contain Dyn). The compilation is written $\Gamma \vdash_{CC} e \rightsquigarrow e' : T$ and it means that e is compiled to e'. Figure 3.2 shows the compilation to cast calculus obtained by *gradualizing* the Simply Typed Lambda Calculus.

3.1.2 Correctness criteria

In [28] is also defined a set of criteria that all gradual type systems must abide and a type and term precision relation (Figure 3.3). The correctness criteria are the following:

Conservative extension: for all static Γ, e and T, $\Gamma \vdash e : T$ iff $\Gamma \vdash_G e : T$
Monotonicity w.r.t. precision: for all Γ, e, e', T, if $\Gamma \vdash_G e : T$ and $e' \sqsubseteq e$, then $\Gamma \vdash_G e' : T'$

$\boxed{\Gamma \vdash_{CC} e \rightsquigarrow e' : T}$ Compilation

$$\frac{x : T \in \Gamma}{\Gamma \vdash_{CC} x \rightsquigarrow x : T} \qquad \frac{\Gamma, x : T_1 \vdash_{CC} e \rightsquigarrow e' : T_2}{\Gamma \vdash_{CC} (\lambda x : T_1 \,.\, e) \rightsquigarrow (\lambda x : T_1 \,.\, e') : T_1 \to T_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{CC} e_1 \rightsquigarrow e_1' : PM_1 \qquad PM_1 \rhd T_1 \to T_2 \\ \Gamma \vdash_{CC} e_2 \rightsquigarrow e_2' : T_1' \qquad T_1' \sim T_1\end{array}}{\Gamma \vdash_{CC} e_1\ e_2 \rightsquigarrow (e_1' : PM_1 \Rightarrow^{l_1} T_1 \to T_2)\ (e_2' : T_1' \Rightarrow^{l_2} T_1) : T_2}$$

Figure 3.2: Compilation to the Cast Calculus

$\boxed{T \sqsubseteq T}$ Type Precision

$$\frac{}{\mathsf{Dyn} \sqsubseteq T} \qquad \frac{}{\mathsf{Bool} \sqsubseteq \mathsf{Bool}} \qquad \frac{T_1 \sqsubseteq T_3 \qquad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4}$$

$\boxed{e \sqsubseteq e}$ Term Precision

$$\frac{}{x \sqsubseteq x} \qquad \frac{T_1 \sqsubseteq T_2 \qquad e_1 \sqsubseteq e_2}{\lambda x : T_1 \,.\, e_1 \sqsubseteq \lambda x : T_2 \,.\, e_2} \qquad \frac{e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{(e_1\ e_2) \sqsubseteq (e_1'\ e_2')}$$

Figure 3.3: Type and Term Precision

and $T' \sqsubseteq T$ for some T'.

Type preservation of cast insertion: for all Γ, e, T, if $\Gamma \vdash_G e : T$, then $\Gamma \vdash e \rightsquigarrow e' : T$ and $\Gamma \vdash_{CC} e' : T$ for some e'.

Monotonicity of cast insertion: for all Γ, e_1, e_2, e_1', T, if $\Gamma \vdash_{CC} e_1 \rightsquigarrow e_1' : T$ and $\Gamma \vdash_{CC} e_2 \rightsquigarrow e_2' : T$ and $e_1 \sqsubseteq e_2$ then $e_1' \sqsubseteq e_2'$.

The first criteria ensures that, for some static program, both type systems (static and gradual) type that program with the same type. The second criteria ensures that adding or removing dynamic type annotations in gradually typed programs does not cause them to become ill-typed. The third and fourth criteria ensures that the cast calculus must be type preserving and monotonic w.r.t. the precision relation.

3.1.3 Methodology

As referred previously, The Gradualizer is composed by a series of steps that ultimately *gradualize* a static type system. In total, there are 6 steps that take on typing rules from a static system and convert those to form typing rules for a gradual system. Each step is applied to all the

individual type rules before moving on to the next step. To generate the compiler to the cast calculus, an additional step is required between steps 5 and 6. In this section, these steps are thoroughly explored.

3.1.3.1 Step 1 - Classify Input/Output Modes

Modes (input/output) deal with predicates that have input parameters (which should be ground before the use of the predicate) and output parameters (which will became ground with the use of the predicate). Typically, in a typing relation $\Gamma \vdash e : T$, the context Γ and the expression e are thought to be inputs while the type T is an output. When classifying, a superscript 'I' or 'O' indicates either input or output mode, respectively. With this notion of input and output modes, each occurrence of types in the typing relation is classified as either being in input or output mode.

Consider the typing rule for application (T-App). When applying this classification, the types that appear on the premises of the type rule are classified as outputs. However, when classifying the conclusion of the type rule, the modes are flipped. This is due to the fact that it is assumed that type T_2 is given as input in the conclusion of the rule.

$$\frac{\Gamma \vdash e_1 : T_1^O \to T_2^O \qquad \Gamma \vdash e_2 : T_1^O}{\Gamma \vdash e_1\ e_2 : T_2^I}\ \text{T-App}$$

3.1.3.2 Step 2 - Classify Producer/Consumer Position

Considering how flows are induced by operational semantics is necessary for designing a gradual system. To approximate this information, each type is classified as either being in a producer or consumer position. Typically, occurrences of types in output modes are classified as producers and occurrences of types in input modes are classified as consumers. When classifying, a superscript 'P' or 'C' indicates either producer or consumer position, respectively.

$$\frac{\Gamma \vdash e_1 : T_1^{OC} \to T_2^{OP} \qquad \Gamma \vdash e_2 : T_1^{OP}}{\Gamma \vdash e_1\ e_2 : T_2^{IC}}\ \text{T-App}$$

This classification is similar to the classification in step 1, with one subtle difference: taking into account the polarity of type constructors. When a type is contravariant, its positions are flipped. In the example above, the arrow type is contravariant in the domain type and covariant in the codomain type, and as such is classified with consumer position in the domain type and with producer position in the codomain type, as shown above.

3.1.3.3 Step 3 - Pattern Match Constructed Outputs

Constructed types are not type variables, but are either base types, such as Bool and Int, or composed by type constructors such as $\to$ (arrow) type and others such as list or sum types

[23]. When a constructed type is in an output position, pattern match is applied to allow the sub-expression to have type Dyn. Consider the rule T-App applied to all previous steps. The type of the expression in the first premise ($T_1^{OC} \to T_2^{OP}$) is a constructed output. Therefore a fresh variable (called a pattern matching variable) is created and the constructed output is replaced by the variable. Then a pattern matching relation between the pattern matching variable and the constructed output is inserted into the premise of the type rule.

$$\frac{\Gamma \vdash e_1 : PM_1 \qquad PM_1 \rhd T_1^{OC} \to T_2^{OP} \qquad \Gamma \vdash e_2 : T_1^{OP}}{\Gamma \vdash e_1\ e_2 : T_2^{IC}}\ \text{T-App}$$

This step handles the case where PM_1 is instantiated with Dyn (considering that $\mathsf{Dyn} \rhd \mathsf{Dyn} \to \mathsf{Dyn}$). This allows less precise programs (that are typed with Dyn) to be well typed. Otherwise, the expression $\lambda f : \mathsf{Dyn}\ .\ \lambda x : \mathsf{Int}\ .\ f\ x$ would not be well typed even though the more precise version $\lambda f : \mathsf{Int} \to \mathsf{Int}\ .\ \lambda x : \mathsf{Int}\ .\ f\ x$ is.

Base types such as Bool and Int also undergo pattern matching, since they are treated as type constructors with arity 0. Consider the typing rule for addition after being applied steps 1 and 2:

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}^{OP} \qquad \Gamma \vdash e_2 : \mathsf{Int}^{OP}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}^{IC}}\ \text{T-Add}$$

As base types are also pattern matched, step 3 produces the resulting type rule (note that occurrences of constructed types in input positions are not pattern matched):

$$\frac{\begin{array}{ll}\Gamma \vdash e_1 : PM_1 & PM_1 \rhd \mathsf{Int}^{OP} \\ \Gamma \vdash e_2 : PM_2 & PM_2 \rhd \mathsf{Int}^{OP}\end{array}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}^{IC}}\ \text{T-Add}$$

This typing rule is equivalent to the one in [14], considering the matching rule for Int:

$$\mathsf{Int} \rhd \mathsf{Int} \qquad \mathsf{Dyn} \rhd \mathsf{Int}$$

3.1.3.4 Step 4 - Flow and Final Type Discovery

Consider the rule for applications after being applied steps 1 through 3:

$$\frac{\Gamma \vdash e_1 : PM_1 \qquad PM_1 \rhd T_1^{OC} \to T_2^{OP} \qquad \Gamma \vdash e_2 : T_1^{OP}}{\Gamma \vdash e_1\ e_2 : T_2^{IC}}\ \text{T-App}$$

First, all variables in output positions are replaced by distinct variables. The two instances of the variable T_1 are replaced with T_1 and T_1', as shown in the rule.

$$\frac{\Gamma \vdash e_1 : PM_1 \qquad PM_1 \rhd T_1^{OC} \to T_2^{OP} \qquad \Gamma \vdash e_2 : T_1'^{OP}}{\Gamma \vdash e_1\ e_2 : T_2^{IC}}\ \text{T-App}$$

Now must be considered how the compilation to the cast calculus will preserve types. For the translation to be well-typed, both T_1 and T_1' must have the same type. As such, we need to choose the final type (the version of the type that will instantiate the type T_1). In order to choose the final type, the following rules are followed in order:

1. If a variable appears in a type annotation, then it is the final type.
2. If a variable has output mode and consumer position, then it is the final type.
3. Otherwise, the final type is the join of all variables in producer position.

In the example above, according to rule 2, T_1 is chosen as final type for T_1 and all it's versions. Now we connect producers to consumers, through the final type, using the $\rightsquigarrow$ (flow) relation. Flow is the same as consistency, however it also guides the insertion of casts. To insert flow information, the following steps are followed:

1. Types in producer position flow to their final types and final types flow to types in consumer position.
2. Variable occurrences in input mode become the final type.

Following these instructions, regarding the example above, the chosen final type is T_1 and the resulting rule is:

$$\frac{\Gamma \vdash e_1 : PM_1 \qquad PM_1 \rhd T_1^{OC} \to T_2^{OP} \qquad \Gamma \vdash e_2 : T_1'^{OP} \qquad T_1' \rightsquigarrow T_1}{\Gamma \vdash e_1\ e_2 : T_2^{IC}}\ \text{T-App}$$

3.1.3.5 Step 5 - Restrict Lone Inputs to Be Static

Lone inputs are variables that appear in input positions only, and therefore are not given a value from output variables. As such, these variables should only range over static types. This step adds this constraint (using the $static(T)$ predicate, which hold only when Dyn does not occur in T) to the typing rule.

Consider the typing rule for abstractions after being applied steps 1 through 4:

$$\frac{\Gamma, x : T_1^{IC} \vdash e : T_2^{OP}}{\Gamma \vdash \lambda x\ .\ e : T_1^{IP} \to T_2^{IC}}\ \text{T-Abs}$$

Without the static constraint, T_1 could range over gradual types and the program $(\lambda x\ .\ x\ x)$ would then be well typed. Therefore the static requirement must be inserted:

$$\frac{\Gamma, x : T_1^{IC} \vdash e : T_2^{OP} \qquad static(T_1)}{\Gamma \vdash \lambda x : T_1^{OP}\ .\ e : T_1^{IP} \to T_2^{IC}}\ \text{T-Abs}$$

3.1.3.6 Step 6 - Replace Flow With Consistency

In the previous steps, flow information is added to aid in producing the gradual type system and the cast insertion rules. Now, that information may be dismissed. All the flows to join types are removed and the remaining flows may be replaced with consistency relations. Considering the recurring example from step 4, the result is:

$$\frac{\Gamma \vdash e_1 : PM_1 \qquad PM_1 \rhd T_1^{OC} \to T_2^{OP} \qquad \Gamma \vdash e_2 : T_1'^{OP} \qquad T_1' \sim T_1}{\Gamma \vdash e_1\ e_2 : T_2^{IC}}\ \text{T-App}$$

3.1.3.7 Step 7 - Generate Casts

This step is applied between steps 5 and 6 in order to generate a cast compiler. As the flows are still present in the type rules, casts will be created according to the flows and pattern matching relations.

Understanding the concept of cast destination is crucial at this step. A cast induced by a pattern matching variable is obtained by expanding the pattern matching variables and by replacing variables according to the flows (taking into account covariance and contravariance of types). Cast destination (for the pattern matching variable) is then this resulting type. In order to generate the compiler, for each expression e add a compilation ($\rightsquigarrow$) relation to e' and:

1. If there is a flow $T \rightsquigarrow T'$, wrap e' in the cast $T \Rightarrow T'$.
2. If there is a pattern matching premise $T \rhd T'$, wrap e' in the cast $T \Rightarrow T''$, where T'' is the cast destination of T.

Returning to the example given previously of the typing rule for application, this step followed by step 6 produces the following rule:

$$\frac{\Gamma \vdash_{CC} e_1 \rightsquigarrow e_1' : PM_1 \qquad PM_1 \rhd T_1^{OC} \to T_2^{OP} \qquad \Gamma \vdash_{CC} e_2 \rightsquigarrow e_2' : T_1'^{OP} \qquad T_1' \sim T_1}{\Gamma \vdash_{CC} e_1\ e_2 \rightsquigarrow (e_1' : PM_1 \Rightarrow^{l_1} T_1 \to T_2)\ (e_2' : T_1' \Rightarrow^{l_2} T_1) : T_2^{IC}}\ \text{T-App}$$

3.1.3.8 Final Step

In this step, the type system (or the compilation to cast calculus) is already completed. Consistency and pattern matching rules need to be added and classifications (mode and position) may be removed. An example of these rules is in Figure 3.1.

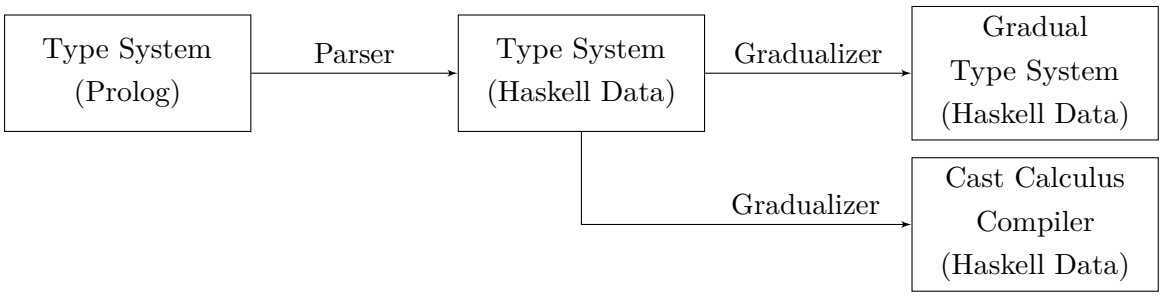


Figure 3.4: Gradualizer Implementation Overview

3.1.4 Implementation

Our implementation of The Gradualizer [9] requires two components in order to produce a gradual type system: a (static) type system written in Prolog and The Gradualizer methodology, which is written in Haskell.

The methodology transforms a static type system into a gradual type system. Therefore it needs static type systems as input so it can produce the corresponding gradual type systems. One example of input of our program is the Simply Typed Lambda Calculus, implemented in Prolog, and presented bellow.

```
-- STLC Signatures
abs :: Term -> Term -> Term
app :: Term -> Term -> Term
arrow :: Type -> Type -> Type


%% Simply Typed Lambda Calculus type system
type(Context, var(X), T) :- member(X:T, Context).
type(Context, abs(X, E), arrow(T1, T2)) :- type([X:T1 | Context], E, T2).
type(Context, app(E1, E2), B) :- type(Context, E1, arrow(A, B)),
                                 type(Context, E2, A).
```

The Simply Typed Lambda Calculus along with several extensions (addition and integers, booleans, exceptions, general recursion, conditional statements, let bindings, recursive let bindings, lists, pairs, references and units) were implemented in Prolog, and the implementation spans over 22 files between type system definitions in Prolog and signature files. A few examples are provided in Appendix A. The implementation is available at https://github.com/pedroangelo/gradualizer.

Along with this dissertation, an implementation of the methodology from the Gradualizer was written in Haskell. Figure 3.4 shows an overview of the implementation. The implementation accepts as input a (static) type system definition in Prolog, parses that input and passes it to The Gradualizer, which then generates the resulting gradual type system and compiler to cast calculus, which are coded using Haskell's Data types. The steps of the methodology are applied to the type system, which is represented using Haskell's Data type.

```
data TypeSystem = TypeSystem [TypeRule]
data TypeRule = TypeRule Premises Conclusion
type Premises = [TypingRelation]
type Conclusion = TypingRelation

data TypingRelation
   = TypeAssignment Context Expression Type
   | MatchingRelation Type Type
   | ConsistencyRelation Type Type
   | StaticRelation Type
   | JoinRelation Type [Type]
   | MemberRelation Bindings Bindings

type Context = [Bindings]
data Bindings = Context String | Binding String Type

data Expression
   = Var String Cast
   | Abstraction String Expression
   | Application Expression Expression
   | Compilation Expression Expression

type Cast = Maybe (Type, Type)
type Name = String

data Type = BaseType Name | VarType Name | DynType | ArrowType Type Type
```

The implementation consists of 3 files for the definition of the type systems and cast calculus, 2 files for parsers, 7 files, one for each step of the methodology, 1 file for examples and 1 file for the top level procedure, totalling at 14 files with roughly 2400 lines of code. The top level of the implementation can be seen in Appendix B. The implementation is available at https://github.com/pedroangelo/gradualizer.

Although in this dissertation we only focus on the Simply Typed Lambda Calculus, the implementation of The Gradualizer supports the following extensions to the STLC: addition and integers, booleans, exceptions, general recursion, conditional statements, let bindings, recursive let bindings, lists, pairs, references and units.

3.2 Type Inference

The constraint generation algorithm is normally designed from a type system, adapting the rules in order to generate constraints. The constraint solving is then built around the existing constraints. The constraint generation described in Figure 2.4 was designed from the STLC, and the constraint unification described in Figure 2.5 was built to solve $\doteq$ (equality) constraints only. Different type systems have different constraint generation algorithms. For example, the Hindley-Milner type system has a type inference algorithm, algorithm W [13], which was built

$\boxed{\Gamma \vdash e : T \mid_{\chi} C}$ Constraint Typing

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\emptyset} \emptyset}\ \text{C-Var}$$

$$\frac{\Gamma, x : X \vdash e : T \mid_{\chi} C}{\Gamma \vdash \lambda x\ .\ e : X \to T \mid_{\chi \cup \{X\}} C}\ \text{C-Abs} \qquad \frac{\Gamma, x : T_1 \vdash e : T_2 \mid_{\chi} C}{\Gamma \vdash \lambda x : T_1\ .\ e : T_1 \to T_2 \mid_{\chi} C}\ \text{C-Abs:}$$

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 : T_1 \mid_{\chi_1} C_1 & \Gamma \vdash e_2 : T_2 \mid_{\chi_2} C_2 \\ cod(T_1) \doteq T_3 \mid_{\chi_3} C_3 & dom(T_1) \dot{\sim} T_2 \mid_{\chi_4} C_4\end{array}}{\Gamma \vdash e_1\ e_2 : T_3 \mid_{\chi_1 \cup \chi_2 \cup \chi_3 \cup \chi_4} C_1 \cup C_2 \cup C_3 \cup C_4}\ \text{C-App}$$

$\boxed{cod(T_1) \doteq T_2 \mid_{\chi} C}$ Constraint Codomain Judgement

$$\frac{}{cod(X) \doteq X_2 \mid_{\{X_1, X_2\}} \{X \doteq X_1 \to X_2\}} \qquad \frac{}{cod(T_1 \to T_2) \doteq T_2 \mid_{\emptyset} \emptyset} \qquad \frac{}{cod(\mathsf{Dyn}) \doteq \mathsf{Dyn} \mid_{\emptyset} \emptyset}$$

$\boxed{dom(T_1) \dot{\sim} T_2 \mid_{\chi} C}$ Constraint Domain Judgement

$$\frac{}{dom(X) \dot{\sim} T_2 \mid_{\{X_1, X_2\}} \{X \doteq X_1 \to X_2, X_1 \dot{\sim} T_2\}} \qquad \frac{}{dom(T_{11} \to T_{12}) \dot{\sim} T_2 \mid_{\emptyset} \{T_{11} \dot{\sim} T_2\}}$$

$$\frac{}{dom(\mathsf{Dyn}) \dot{\sim} T_2 \mid_{\emptyset} \{\mathsf{Dyn} \dot{\sim} T_2\}}$$

Figure 3.5: Constraint Generation

according to the rules of the Hindley-Milner type system. However, as the type system only uses $\doteq$ (equality) constraints, the constraint unification algorithm used in Damas-Milner is first order unification as presented in Figure 2.5. Gradual typing introduces the Dyn (dynamic) type and $\sim$ (consistency) relation. As such, a type inference algorithm that can deal with these new additions is required. Type inference for gradual typing was presented in [14].

3.2.1 Constraint Generation

Constraint generation as presented in [14] (shown in Figure 3.5) deals with the Dyn (dynamic) type, by introducing the *cod* and *dom* relations. *cod* is responsible for retrieving the codomain of a type while *dom* is responsible for retrieving the domain of a type, and both also produce the necessary constraints to associate new type variables with types. These relations allow for a different treatment of the Dyn (dynamic) type.

Consider the expression given as an example in Chapter 1:

$$(\lambda x : \mathsf{Dyn} \;.\; 1 + x)\ \mathsf{true}$$

Some extra constraint typing judgements [14] are necessary to generate constraints for this expression:

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool} \mid_{\emptyset} \emptyset}\ \text{C-True} \qquad \frac{}{\Gamma \vdash n : \mathsf{Int} \mid_{\emptyset} \emptyset}\ \text{C-Int}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid_{\chi_1} C_1 \qquad \Gamma \vdash e_2 : T_2 \mid_{\chi_2} C_2}{\Gamma \vdash e_1 + e_2 : \mathsf{Int} \mid_{\chi_1 \cup \chi_2} C_1 \cup C_2 \cup \{T_1 \dot{\sim} \mathsf{Int}, T_2 \dot{\sim} \mathsf{Int}\}}\ \text{C-Add}$$

It was already demonstrated that this expression is typeable and has type Int. The constraint generation is exemplified bellow. For the sake of simplicity lets elide χ variables.

$$1\ \frac{\dfrac{\dfrac{}{x : \mathsf{Dyn} \vdash 1 : \mathsf{Int} \mid \emptyset}\ \text{C-Int} \qquad \dfrac{}{x : \mathsf{Dyn} \vdash x : \mathsf{Dyn} \mid \emptyset}\ \text{C-Var}}{x : \mathsf{Dyn} \vdash 1 + x : \mathsf{Int} \mid \{\mathsf{Int} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Int}\}}\ \text{C-Add}}{\vdash \lambda x : \mathsf{Dyn} \;.\; 1 + x : \mathsf{Dyn} \to \mathsf{Int} \mid C_1 \triangleq \{\mathsf{Int} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Int}\}}\ \text{C-Abs:}$$

$$\frac{1\ \dfrac{}{\vdash \lambda x : \mathsf{Dyn} \;.\; 1 + x : \mathsf{Dyn} \to \mathsf{Int} \mid C_1}\ \text{C-Abs:} \qquad \dfrac{}{\vdash \mathsf{true} : \mathsf{Bool} \mid \emptyset}\ \text{C-True} \qquad \mathit{cod}(\mathsf{Dyn} \to \mathsf{Int}) \doteq \mathsf{Int} \mid \emptyset \qquad \mathit{dom}(\mathsf{Dyn} \to \mathsf{Int}) \dot{\sim} \mathsf{Bool} \mid \{\mathsf{Dyn} \dot{\sim} \mathsf{Bool}\}}{\vdash (\lambda x : \mathsf{Dyn} \;.\; 1 + x)\ \mathsf{true} : \mathsf{Int} \mid \{\mathsf{Int} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Bool}\}}\ \text{C-App}$$

This expression has type Int and the constraints generated are $\{\mathsf{Int} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Bool}\}$.

3.2.2 Constraint Solving

In [14], constraint solving (shown in Figure 3.6) is extended with new rules that deal with $\dot{\sim}$ (consistency) constraints. The rules are divided into two groups, those that solve $\doteq$ (equality) constraints and those that solve $\dot{\sim}$ (consistency) constraints. The rules that solve $\doteq$ (equality) constraints are similar to those in Figure 2.5 and focus on comparing static types, ensuring the expression is well typed. The rules that solve $\dot{\sim}$ (consistency) constraints focus on comparing gradual types, reducing to $\doteq$ (equality) the $\dot{\sim}$ (consistency) constraints between static types, and otherwise discarding $\dot{\sim}$ (consistency) constraints between the Dyn (dynamic) type and some other type. The constraint solving rules for gradual typing are somewhat different from the usual constraint solving rules. The constraint solving procedure $\tau \mid C \upsilon S$ solves (υ) a set of constraints C, resulting in a set of substitutions S and a set of gradual types τ. $BType$ is the set where all base types belong to, $TVar$ is the set where all type variables belong to and $TParam$ is the set where all type parameters (base types with unknown identity) belong to. Like in Figure 2.5, X represents type variables, T represents types and $Vars(T)$ represents the set of all type variables in T.

$\boxed{\tau \mid C \upsilon S}$ Constraint Solving

$$\frac{}{\emptyset \upsilon S}$$

$$\frac{\tau \mid C \upsilon S \qquad T \notin BType \cup TVar \cup TParam}{\tau \mid C \cup \{T \dot{\sim} T\} \upsilon S} \qquad \frac{\tau \cup \{T\} \mid C \upsilon S}{\tau \mid C \cup \{\mathsf{Dyn} \dot{\sim} T\} \upsilon S}$$

$$\frac{\tau \cup \{T\} \mid C \upsilon S}{\tau \mid C \cup \{T \dot{\sim} \mathsf{Dyn}\} \upsilon S} \qquad \frac{\tau \mid C \cup \{T_{11} \dot{\sim} T_{21}, T_{12} \dot{\sim} T_{22}\} \upsilon S}{\tau \mid C \cup \{T_{12} \to T_{12} \dot{\sim} T_{21} \to T_{22}\} \upsilon S}$$

$$\frac{\tau \mid C \cup \{X \dot{\sim} T\} \upsilon S \qquad T \notin TVar}{\tau \mid C \cup \{T \dot{\sim} X\} \upsilon S}$$

$$\frac{\tau \mid C \cup \{X \doteq T\} \upsilon S \qquad T \notin BType \cup TVar \cup TParam}{\tau \mid C \cup \{X \dot{\sim} T\} \upsilon S}$$

$$\frac{\begin{array}{c}\{X_1, X_2\} fresh \qquad X \notin Vars(T_1 \to T_2) \\ \tau \mid C \cup \{X \doteq X_1 \to X_2, X_1 \dot{\sim} T_1, X_2 \dot{\sim} T_2\} \upsilon S\end{array}}{\tau \mid C \cup \{X \dot{\sim} T_1 \to T_2\} \upsilon S}$$

$$\frac{\tau \mid C \upsilon S \qquad T \notin BType \cup TVar \cup TParam}{\tau \mid C \cup \{T \doteq T\} \upsilon S} \qquad \frac{\tau \mid C \cup \{T_{11} \doteq T_{21}, T_{12} \doteq T_{22}\} \upsilon S}{\tau \mid C \cup \{T_{12} \to T_{12} \doteq T_{21} \to T_{22}\} \upsilon S}$$

$$\frac{\tau \mid C \ \cup \{X \doteq T\} \upsilon S \qquad T \notin TVar}{\tau \mid C \cup \{T \doteq X\} \upsilon S} \qquad \frac{[X \mapsto T](\tau) \mid [X \mapsto T](C) \upsilon S \qquad X \notin Vars(T)}{\tau \mid C \cup \{X \doteq T\} \upsilon S \circ [X \mapsto T]}$$

Figure 3.6: Constraint Solving

Considering the previous example, the constraints generated must now be solved to generate substitutions S and gradual types τ.

$$\frac{\frac{\frac{\frac{}{\{Bool, Int\} \mid \emptyset \upsilon \emptyset}}{\{Bool, Int\} \mid \{\mathsf{Int} \dot{\sim} \mathsf{Int}\} \upsilon \emptyset}}{\{Bool\} \mid \{\mathsf{Int} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Int}\} \upsilon \emptyset}}{\emptyset \mid \{\mathsf{Int} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Int}, \mathsf{Dyn} \dot{\sim} \mathsf{Bool}\} \upsilon \emptyset}$$

The constraint solving produced the substitutions $\emptyset$ and gradual types $\{Bool, Int\}$. Now, for each free variable in the gradual types, a substitution from that variable to the dynamic type must be added to the substitutions S. This is required because there may be type variables that are consistent with the dynamic type and have no other constraints. Since constraints between

some type and the dynamic type are ignored, these type variables must be collected and then replaced with the dynamic type. Finally, the substitutions must be applied to the final type. As there are no substitutions, the expression $(\lambda x : \mathsf{Dyn}\ .\ 1 + x)\ \mathsf{true}$ is accepted and has final type Int.

3.2.3 Implementation

We implemented the type inference algorithm for gradual typing in Haskell. The implementation is integrated as a module of the interpreter for the gradual language, which we will discuss in Chapter 4.

The implementation accepts expressions (programs) which are stored using Haskell's Data types.

```
data Expression
   = Variable Var
   | Abstraction Var Expression
   | Application Expression Expression
   | Annotation Var Type Expression

data Type
   = VarType Var
   | ArrowType Type Type
   | DynType
```

An overview of the type inference implementation is shown in Figure 3.7. The expression is passed to the type inference algorithm which is divided in two main phases: constraint generation and constraint solving. The program goes through constraint generation, depicted as an arrow labelled as CG, which produces the type of the expression, constraints and a fully annotated version of the expression. Then the constraints are passed to the constraint solver, depicted as an arrow with the label CS, which produces substitutions. These substitutions are then applied to the type of the expression to produce the final type of the expression and to each annotation in the annotated expression. The annotated expression will be useful during cast insertion.

To show the application of the type inference algorithm, consider the following expression:

$$(\lambda x : \mathsf{Dyn}\ .\ \mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 2)\ ((\lambda x : \mathsf{Dyn}\ .\ (\lambda y\ .\ y)\ x)\ \mathsf{true})$$

This expressions is encoded in our implementation as:

```
*TypeInference> parameters_6
Application (Annotation "x" DynType (If (Variable "x") (Int 1) (Int 2)))
    (Application (Annotation "x" DynType (Application (Abstraction "y"
    (Variable "y")) (Variable "x"))) (Bool True))
```

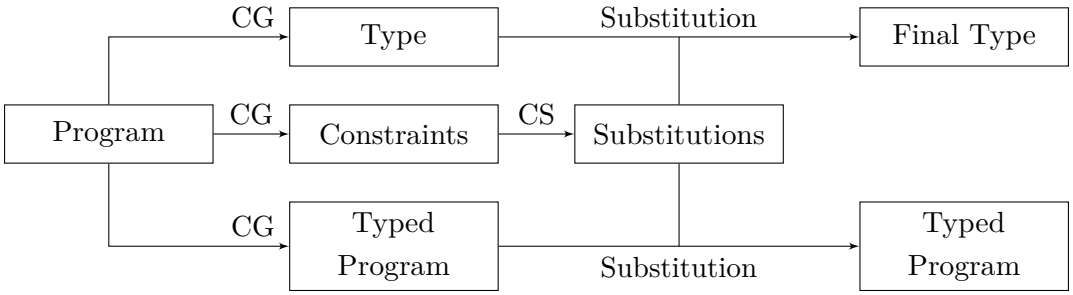


Figure 3.7: Type Inference Implementation Overview

The first phase of type inference is constraint generation, which either returns type errors or the type of the expression, as well as constraints for typeability. In our implementation, constraint generation also returns expression with type annotations in each term.

```
*TypeInference> let ta = ([], parameters_6)
*TypeInference> let Right ((typ, constraints, expr_typed), counter) = runExcept
    $ runStateT (generateConstraints ta) 1
*TypeInference> typ
IntType
*TypeInference> constraints
[Consistency DynType BoolType,Consistency (VarType "t2") DynType,Consistency
    DynType BoolType,Consistency DynType (VarType "t2")]
*TypeInference> expr_typed
TypeInformation IntType (Application (TypeInformation (ArrowType DynType
    IntType) (Annotation "x" DynType (TypeInformation IntType (If
    (TypeInformation DynType (Variable "x")) (TypeInformation IntType (Int 1))
    (TypeInformation IntType (Int 2)))))) (TypeInformation (VarType "t2")
    (Application (TypeInformation (ArrowType DynType (VarType "t2"))
    (Annotation "x" DynType (TypeInformation (VarType "t2") (Application
    (TypeInformation (ArrowType (VarType "t2") (VarType "t2")) (Abstraction "y"
    (TypeInformation (VarType "t2") (Variable "y")))) (TypeInformation DynType
    (Variable "x")))))) (TypeInformation BoolType (Bool True)))))
```

Then, the constraints need to be solved. Solving constraints will result in either a type error or if no static type errors are present, a set of gradual types and a set of substitutions:

```
*TypeInference> let Right (gtypes, substitutions) = runExcept $
    unifyConstraints [] (reverse constraints) counter
*TypeInference> gtypes
[VarType "t2",BoolType,VarType "t2",BoolType]
*TypeInference> substitutions
[]
```

For each type variable contained in gradual types, a substitution from that type variable to the dynamic type must be added to the substitutions:

```
*TypeInference> let gtypes' = nub $ concat $ map (\x -> map (VarType) $
```

```
    freeVariables x) gtypes
*TypeInference> let substitutions' = (map (\x -> (x, DynType)) gtypes') ++
    substitutions
*TypeInference> substitutions'
[(VarType "t2",DynType)]
```

Finally, the substitutions are applied to the types in order to infer the correct type for the expression:

```
*TypeInference> let finalType = foldr substituteType typ substitutions'
*TypeInference> finalType
IntType
```

The substitutions must also be applied to each type in the fully annotated expression:

```
*TypeInference> let typedExpr = substituteTypedExpression substitutions'
    expr_typed
*TypeInference> typedExpr
TypeInformation IntType (Application (TypeInformation (ArrowType DynType
    IntType) (Annotation "x" DynType (TypeInformation IntType (If
    (TypeInformation DynType (Variable "x")) (TypeInformation IntType (Int 1))
    (TypeInformation IntType (Int 2)))))) (TypeInformation DynType (Application
    (TypeInformation (ArrowType DynType DynType) (Annotation "x" DynType
    (TypeInformation DynType (Application (TypeInformation (ArrowType DynType
    DynType) (Abstraction "y" (TypeInformation DynType (Variable "y"))))
    (TypeInformation DynType (Variable "x")))))) (TypeInformation BoolType
    (Bool True)))))
```

Our implementation consists of 1 file for constraint generation, 1 file for constraint solving and 1 file for the type inference procedure, totalling at 3 files with roughly 1400 lines of Haskell code. The top level of the implementation can be seen in Figure 4.6. The implementation is available at https://github.com/pedroangelo/interpreter.

Although in this dissertation we only focus on the Gradually Typed Lambda Calculus, the implementation of the type inference algorithm also supports the following extensions to the STLC: integers, booleans, let bindings, general recursion, recursive let bindings, conditional statements, arithmetic operations, comparison operations between integers, units, pairs, tuples, records, sums, variants, lists and recursive types.

Chapter 4

Operational Semantics

4.1 Gradualizer

The semantics for the Cast Calculus (Figure 4.1) is introduced in [10] through a methodology (similar to [9]) that automatically derives the reduction rules from a statically typed system's reduction rules. These rules allow casts to be reduced, and thus allow the evaluation of the Cast Calculus.

There are three different sets of evaluations rules that are used to evaluate expressions in the Cast Calculus. The evaluation rules from the STLC, namely E-App1, E-App2 and E-AppAbs are used to evaluate expressions without casts. These rules came from the STLC and are incorporated into the CC. The evaluation rules present in Figure 4.1, namely ID-BASE, and the rules that are derived from a static system through the methodology, such as C-BETA, are specific to the CC. The cast handler reduction rules, in Figure 4.2, are independent of the language and pertain casts.

Unlike in Chapter 3, the analogous methodology to The Gradualizer [9], whose purpose is to automatically generate the reduction rules [10], is not thoroughly explained here, neither was implemented. Instead, we implemented the evaluation rules directly on the interpreter for the gradually typed language.

4.1.1 Reduction Rules

The reduction rules from the STLC are necessary because they enable the reduction of the original terms. The reduction rules present in Figure 4.1 are used to reduce cast expressions between the different type constructors that the static system may support. A language with product ($\times$), sum ($+$) or list types needs cast reduction rules for reducing casts with these type constructors.

However, the cast handler reduction rules are somewhat different. These rules allow types to be checked during runtime and *blame errors* to form when these types are not consistent.

Syntax

$$
\begin{array}{llll}
\textit{Expressions} & e & ::= & x \mid \lambda x : T \,.\, e \mid e\ e \mid e : T \Rightarrow^{l} T \mid \mathsf{blame}_T\ l \\
\textit{Values} & v & ::= & \lambda x : T \,.\, e \mid \mathsf{blame}_T\ l \mid v : G \Rightarrow^{l} \mathsf{Dyn} \mid v : T_1 \to T_2 \Rightarrow^{l} T_1' \to T_2' \\
\textit{Types} & T & ::= & B \mid \mathsf{Dyn} \mid T \to T \\
\textit{Ground Types} & G & ::= & B \mid \mathsf{Dyn} \to \mathsf{Dyn} \\
\textit{Base Types} & B & ::= & \mathsf{Bool} \mid \mathsf{Int} \\
\textit{Context} & \Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}
$$

$\boxed{\Gamma \vdash_G e : T}$ Typing

$$\frac{\Gamma \vdash_G e : T_1 \qquad T_1 \sim T_2}{\Gamma \vdash_G (e : T_1 \Rightarrow^{l} T_2) : T_2}\ \text{T-Cast} \qquad \frac{}{\Gamma \vdash_G \mathsf{blame}_T\ l : T}\ \text{T-Blame}$$

$\boxed{T \sim T}$ Consistency

$$\frac{}{B \sim B} \qquad \frac{}{T \sim \mathsf{Dyn}} \qquad \frac{}{\mathsf{Dyn} \sim T} \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$$

$\boxed{e \longrightarrow e}$ Evaluation

rules in Figure 4.2 and

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}\ \text{E-App1} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'}\ \text{E-App2} \qquad \frac{}{(\lambda x : T \,.\, e)\ v \longrightarrow [x \mapsto v]e}\ \text{E-AppAbs}$$

$$\frac{e \longrightarrow v}{e : T_1 \Rightarrow^{l} T_2 \longrightarrow v : T_1 \Rightarrow^{l} T_2}\ \text{E-Cast} \qquad \frac{}{v : B \Rightarrow^{l} B \longrightarrow v}\ \text{ID-BASE}$$

$$\frac{}{(v_1 : T_1 \to T_2 \Rightarrow^{l} T_3 \to T_4)\ v_2 \longrightarrow (v_1\ (v_2 : T_3 \Rightarrow^{l} T_1)) : T_2 \Rightarrow^{l} T_4}\ \text{C-BETA}$$

Figure 4.1: Cast Calculus (CC)

Rule ID-STAR reduces the identity cast on Dyn, which is a cast whose source and target is the dynamic type. Rule SUCCEED handles a cast from the same ground type through the dynamic type, which is basically a successful type check. This rule leaves the value unchanged and removes the casts. Rule FAIL, on the other hand, handles the casts from a ground type to a different ground type through the dynamic type. This cast is a failed type check, and therefore produces a *blame error*. The rules GROUND and EXPAND allow type constructor casts to be factored through their ground types.

$$v : \mathsf{Dyn} \Rightarrow^{l} \mathsf{Dyn} \longrightarrow v \qquad \text{(ID-STAR)}$$

$$v : G \Rightarrow^{l_1} \mathsf{Dyn} : \mathsf{Dyn} \Rightarrow^{l_2} G \longrightarrow v \qquad \text{(SUCCEED)}$$

$$v : G_1 \Rightarrow^{l_1} \mathsf{Dyn} : \mathsf{Dyn} \Rightarrow^{l_2} G_2 \longrightarrow \mathsf{blame}_{G_2}\ l_2 \quad if\ G_1 \neq G_2 \qquad \text{(FAIL)}$$

$$v : T \Rightarrow^{l} \mathsf{Dyn} \longrightarrow v : T \Rightarrow^{l} G : G \Rightarrow^{l} \mathsf{Dyn} \quad if\ T \neq \mathsf{Dyn}, T \neq G, T \sim G \qquad \text{(GROUND)}$$

$$v : \mathsf{Dyn} \Rightarrow^{l} T \longrightarrow v : \mathsf{Dyn} \Rightarrow^{l} G : G \Rightarrow^{l} T \quad if\ T \neq \mathsf{Dyn}, T \neq G, T \sim G \qquad \text{(EXPAND)}$$

$$\frac{\emptyset \vdash E[\mathsf{blame}_{T_1}\ l] : T_2}{E[\mathsf{blame}_{T_1}\ l] \longrightarrow \mathsf{blame}_{T_2}\ l}\ \text{CTX-BLAME}$$

Figure 4.2: Cast Handler Reduction Rules

4.1.2 Correctness Criteria

Like the type system, the cast language derived from the methodology must also satisfy certain correctness criteria:

Conservative extension: for all static e and e', $e \longrightarrow_s e'$ iff $e \longrightarrow e'$.
Type Safety: both progress and type preservation hold.
Blame Theorem: for all typed e, for all T and l, $e \longrightarrow^{\star} \mathsf{blame}_T\ l$ implies that l is not in a safe cast of e.
Gradual Guarantee: for all typed $e_1 \sqsubseteq e_2$, (1) if $e_2 \longrightarrow e_2'$ then $e_1 \longrightarrow^{\star} e_1'$ and $e_1' \sqsubseteq e_2'$. (2) if $e_1 \longrightarrow e_1'$ then either $e_2 \longrightarrow^{\star} e_2'$ and $e_1' \sqsubseteq e_2'$, or $e_2 \longrightarrow^{\star} \mathsf{blame}_T\ l$.

The first criteria ensures that a static program evaluates to the same result regardless of being evaluated in the STLC or in the Cast Calculus. The second criteria ensures that types are preserved as the expressions are reduced. The third criteria states that statically typed code can only introduce safe casts in the cast calculus, thus ensuring that statically typed code is never blamed. In the fourth criteria, part (1) ensures that removing type annotations from a program that does not crash and thus returns a value, cannot make it crash or return a different value. Part (2) states that by adding type annotations to a program that returns a value, that program can either return the same value or a *blame error*.

4.2 A Gradually Typed Programming Language

The Gradualizer system [9, 10], seen so far in Chapter 3 and in Chapter 4, aims to derive type system, cast insertion and evaluation rules for gradual typing. These rules form some of the necessary components to implement a language with gradual typing. Instead of having type declarations, the compiler infers types, and therefore it needs a type inference algorithm, which

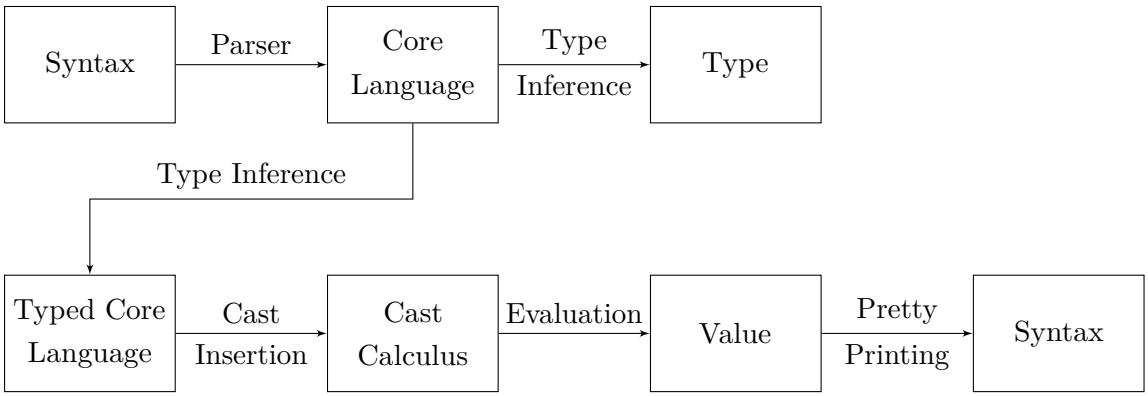


Figure 4.3: Interpreter Implementation Overview

was also covered in Chapter 3.

These components form the basis for implementing a gradually typed language, consisting of static and dynamic type checking along with reduction rules for evaluating terms. There are other components who are also very useful to an implementation, particularly a parser and pretty printer, which allows a programmer to write and read code in clear syntax, instead of dealing with machine code.

The execution of a programming language is usually divided in two phases: compilation and evaluation. Compilation is the phase that takes the program written in a file and compiles it into executable machine code. The evaluation phase then takes this code and executes it, producing a value or halting execution due to an exception or error. In Figure 4.3 is an overview of the different phases of the interpreter. The compilation phase of the implementation is divided in the following steps: first the program is parsed, and translated into intermediate machine code known as core language; then the type inference algorithm will type check the program; if it type checks, the type inference will pass that intermediate code to the cast insertion algorithm, if it does not type check, a type error is thrown; the cast insertion will produce the cast calculus, which is the core language expanded with casts. The evaluation phase is composed of the following steps: the compilation produced the cast calculus, which is then evaluated, either producing the resulting value or a *blame error*; then the result is pretty printed and shown to the user.

4.2.1 Syntax

The first step in defining a language is to define its syntax. The syntax definition should be clear and simple, and it is what sets the keywords and symbols with which programs can be expressed. It also facilitates the writing of programs, that otherwise would have to be written in intermediate machine code. The full syntax definition for the implementation is presented in Figures 4.4 and 4.5.

The language features the Gradually Typed Lambda Calculus along with many (*gradualized*)

Expressions e	$::= var$	variable
	$\mid \lambda var\ .\ e$	lambda abstraction
	$\mid e\ e$	application
	$\mid e :: T$	ascription
	$\mid \lambda var : T\ .\ e$	annotated lambda abstraction
	$\mid n$	integer
	$\mid$ true	boolean true
	$\mid$ false	boolean false
	$\mid$ let $var = e$ in e	let binding
	$\mid$ fix e	fixed point
	$\mid$ letrec $var = e$ in e	recursive let binding
	$\mid$ if e then e else e	conditional statement
	$\mid e\ aop\ e$	arithmetic operation
	$\mid e\ rop\ e$	relational operation
	$\mid$ unit	Unit
	$\mid (tuple)$	tuple
	$\mid$ proj $n\ n\ e$	tuple projection
	$\mid \{record\}$	record
	$\mid e.var$ as T	record projection
	$\mid$ case e of $alternatives$	case
	$\mid \langle label = e \rangle$ as T	tag
	$\mid$ nil $[T]$	empty list (nil)
	$\mid []$ as T	empty list
	$\mid$ cons $[T]\ e\ e$	list constructor
	$\mid e : e$ as T	list constructor operator
	$\mid [list]$ as T	list
	$\mid$ isnil $[T]\ e$	test for empty list
	$\mid$ head $[T]\ e$	head of a list
	$\mid$ tail$[T]\ e$	tail of a list
	$\mid$ fold $[T]\ e$	folding
	$\mid$ unfold $[T]\ e$	unfolding

Figure 4.4: Syntax

$$
\begin{array}{rcll}
aop & ::= & + \mid - \mid * \mid / & \text{arithmetic operators} \\
rop & ::= & == \mid /= \mid < \mid > \mid <= \mid >= & \text{relational operators} \\
tuple & ::= & e,\ e \mid e,\ tuple & \\
record & ::= & var = e \mid var = e,\ record & \\
alternatives & ::= & \mid label\ var \to e & \text{alternatives} \\
 & & \mid \mid label\ var \to e,\ alternatives & \\
list & ::= & e \mid e,\ list & \text{list} \\
Types\ T & ::= & var & \text{type variable} \\
 & & \mid T \to T & \text{arrow type} \\
 & & \mid \mathsf{Int} & \text{integer type} \\
 & & \mid \mathsf{Bool} & \text{boolean type} \\
 & & \mid \mathsf{Dyn} & \text{dynamic type} \\
 & & \mid \mathsf{Unit} & \text{unit type} \\
 & & \mid \mathsf{rec}\ var\ .\ T & \text{recursive type} \\
 & & \mid (tupleType) & \text{tuple type} \\
 & & \mid \{recordType\} & \text{record type} \\
 & & \mid \langle variantType \rangle & \text{variant type} \\
 & & \mid [T] & \text{list type} \\
tupleType & ::= & T, T \mid T,\ tupleType & \\
recordType & ::= & var : T \mid var : T,\ recordType & \\
variantType & ::= & label : T \mid label : T,\ variantType &
\end{array}
$$

Figure 4.5: Syntax (cont.)

STLC extensions. We chose not to include products and sums because these are base cases for tuples and variants, which the language already supports.

Some objects in the syntax requires additional explanation. *var* represents a variable, which can be any string starting with a lower case alphabetic character followed by alphanumeric characters, an underscore or quotation mark. n represents a positive integer. *label* represents a string starting with an upper case alphabetic character followed by alphanumeric characters, an underscore or quotation mark.

```
-- type inference procedure
typeInference :: Expression -> Either String (Type, Expression)
typeInference expr = do
   -- build type assignment from expression and expression
   let ta = ([], expr)
   -- generate constraints
   cg <- runExcept $ runStateT (generateConstraints ta) 1
   -- retrieve constraints
   let ((typ, constraints, expr_typed), counter) = cg
   -- unify constraints and generate substitutions
   cu <- runExcept $ unifyConstraints [] (reverse constraints) counter
   -- retrieve gradual types and substitutions
   let (gtypes, substitutions) = cu
   -- filter gradual type variables
   let gtypes' = nub $ concat $
      map (\x -> map (VarType) $ freeVariables x) gtypes
   -- add substitutions from types present in gtypes to dynamic type
   let substitutions' = (map (\x -> (x, DynType)) gtypes') ++ substitutions
   -- replace unconstrained type variables by type parameters
   -- discover final type by applying all substitutions to expression type t
   let finalType = foldr substituteType typ substitutions'
   -- replace unconstrained type variables by type parameters
   -- discover final types by applying all substitutions to each type
       ascription and type information in the expression
   let typedExpr = substituteTypedExpression substitutions' expr_typed
   return (finalType, typedExpr)
```

Figure 4.6: Type Inference

4.2.2 Parsing

Our parser is written using Haskell's Parsec library [19]. Part of the implementation of the parser is presented in Appendix C.

4.2.3 Type Inference

The next step is to type check the program to make sure it does not have errors. Instead of type checking, the language infers types using a type inference algorithm. As the language has support for gradual types and let-polymorphism, the type inference algorithm uses ideas of gradual typing [9, 14], detailed in Chapter 3, algorithm W [13] and HM(X) [21], detailed in Chapter 2. The top level of the type inference is presented in Figure 4.6.

4.2.4 Cast Insertion

If the program contains no type errors, the type inference algorithm will produce the type of the program along with a fully type-annotated version of the program that will be passed to the cast insertion [9] procedure. In this step, casts will be added to the program to ensure type errors are caught during runtime. The result of cast insertion is called the cast calculus. The rules for cast insertion are presented in Chapter 3.

4.2.5 Evaluation

Now that the program has been compiled to the cast calculus, it can be evaluated. Evaluation is done using the rules introduced in [10], which are specified in Chapter 4. If there is some type error that was not caught during type checking, it will be caught at this phase, and a *blame error* will be produced. If there are no type errors, the evaluation rules will evaluate the program and produce the resulting value.

4.2.6 Pretty Printing

Finally, once the program has been evaluated and it has produced a result, that result is printed using a pretty printer, which converts the intermediate code into syntax in a presentable and readable manner, formatting the code so that it retains indentations. In the implementation of the pretty printer, we used Haskell's Text.PrettyPrint.Leijen library for pretty printing combinators. This library was based on pretty printing combinators described in [33].

4.2.7 Implementation

To demonstrate these phases in the interpreter, consider the following expression:

$$(\lambda x : \mathsf{Dyn}\ .\ \mathsf{if}\ x\ \mathsf{then}\ 1\ \mathsf{else}\ 2)\ ((\lambda x : \mathsf{Dyn}\ .\ (\lambda y\ .\ y)\ x)\ \mathsf{true})$$

When parsed, this expression is converted to the following intermediate code:

```
*Interpreter> parse expressionParser "" "(\\x : Dyn . if x then 1 else 2) ((\\x
    : Dyn . (\\y . y) x) true)"
Right (Application (Annotation "x" DynType (If (Variable "x") (Int 1) (Int 2)))
    (Application (Annotation "x" DynType (Application (Abstraction "y"
    (Variable "y")) (Variable "x"))) (Bool True)))
```

Now the expression must have its type inferred. If a type error is found, the expression is not evaluated.

```
*Interpreter> inferType t
Right IntType
```

The expression has type Int, therefore it can be evaluated. Next step is to annotate, with their types, each term of the expression:

```
*Interpreter> let Right annotatedExpression = insertTypeInformation t
*Interpreter> annotatedExpression
TypeInformation IntType (Application (TypeInformation (ArrowType DynType
    IntType) (Annotation "x" DynType (TypeInformation IntType (If
    (TypeInformation DynType (Variable "x")) (TypeInformation IntType (Int 1))
    (TypeInformation IntType (Int 2)))))) (TypeInformation DynType (Application
    (TypeInformation (ArrowType DynType DynType) (Annotation "x" DynType
    (TypeInformation DynType (Application (TypeInformation (ArrowType DynType
    DynType) (Abstraction "y" (TypeInformation DynType (Variable "y"))))
    (TypeInformation DynType (Variable "x")))))) (TypeInformation BoolType
    (Bool True)))))
```

With the expression fully annotated, casts, that will catch runtime errors, must be inserted:

```
*Interpreter> let castExpression = removeTypeInformation $ insertCasts
    annotatedExpression
*Interpreter> castExpression
Application (Cast (ArrowType DynType IntType) (ArrowType DynType IntType)
    (Annotation "x" DynType (If (Cast DynType BoolType (Variable "x")) (Cast
    IntType IntType (Int 1)) (Cast IntType IntType (Int 2))))) (Cast DynType
    DynType (Application (Cast (ArrowType DynType DynType) (ArrowType DynType
    DynType) (Annotation "x" DynType (Application (Cast (ArrowType DynType
    DynType) (ArrowType DynType DynType) (Abstraction "y" (Variable "y")))
    (Cast DynType DynType (Variable "x"))))) (Cast BoolType DynType (Bool
    True))))
```

Once casts have been inserted, the expression is ready to be evaluated:

```
*Interpreter> let result = evaluate castExpression
*Interpreter> result
Int 1
```

The result of evaluating the expression is 1. Now to show the programmer the result, we pretty print it:

```
*Interpreter> prettyExpression result
1
```

The implementation for the interpreter was written is Haskell and is composed of several phases, all explained above. The implementation consists of 1 file for the parser definition, 2 files for the language definition (terms and types), 3 files for type inference (top level, constraint generation and solving), 1 file for cast insertion rules, 1 file for evaluation rules, 1 file for pretty printing, 1 file with examples and the file for the interpreter top level. In total there are 10 files with roughly 5300 lines of Haskell code. The top level of the interpreter can be seen in Appendix D. The implementation is available at https://github.com/pedroangelo/interpreter.

Chapter 5

Gradual Data Types

One useful addition to gradual typing is to allow the definition of algebraic data types (such as Data in Haskell) with elements of type Dyn (gradual data types). To implement data types, combination and alternation will have to be simulated using products and sums or their generalizations, tuples and variants. However, several obstacles prevent the implementation of gradual data types. The research presented in this chapter is an original contribution [2].

5.1 Obstacles to Gradual Data types

Consider the data type definition (Haskell syntax) of the instance of *Maybe* with elements of type Int:

$$data\ MaybeInt = Nothing \mid Just\ Int$$

Using algebraic type constructors [23], we can represent *MaybeInt* as the type Unit + Int using a sum type. We may use the standard operations over sum types, such as case, inl and inr to build expressions that interact with values of type *MaybeInt*, define values of type *MaybeInt* with explicit type annotations and apply the expressions to the values. To implement the instance of *Maybe* with elements of type Dyn,

$$data\ MaybeDyn = Nothing \mid Just\ Dyn$$

one would assume that all that is required is to replace Int with Dyn in the type annotations in the expressions. However, the type Dyn is only allowed in type annotations in abstractions under the type system from [9] that is presented in Chapter 3.

Now consider the following example of a list of integers:

$$data\ ListInt = Nil \mid Cons\ Int\ ListInt$$

Here we have two types (*Int* and *ListInt*) in the alternative on the right. Product types are used to represent the combination of *Int* and *ListInt*, thus allowing the definition of data types with

more than one element. Product types have 3 standard terms: the pair, fst and snd. However, there is no way to annotate the pair term with types. Therefore, we could not specify that a certain element of the product type has type Dyn.

Despite the fact that the type system does not allow the type Dyn in type annotations, except those in abstractions, and the fact that pairs don't have type annotations that would allow the specification of dynamic elements in data types, there is another obstacle. Even if the type system could support dynamic type annotations besides those in abstractions and the syntax of pairs allowed type annotations, the cast insertion would not introduce the necessary casts that ensure a certain element is treated dynamically. Consider the following example:

$$(\lambda x : \mathsf{Dyn} \;.\; 1 + x)\ \mathsf{true}$$

This expression type checks and admits type Int because the lambda abstraction has type $\mathsf{Dyn} \to \mathsf{Int}$ and true has type Bool which is admissible for a dynamically typed argument. The cast insertion procedure produces the expression[1]:

$$(\lambda x : \mathsf{Dyn} \;.\; 1 + (x : \mathsf{Dyn} \Rightarrow \mathsf{Int}))\ (\mathsf{true} : \mathsf{Bool} \Rightarrow \mathsf{Dyn})$$

which evaluates to

$$1 + ((\mathsf{true} : \mathsf{Bool} \Rightarrow \mathsf{Dyn}) : \mathsf{Dyn} \Rightarrow \mathit{Int})$$

The cast $((\mathsf{true} : \mathsf{Bool} \Rightarrow \mathsf{Dyn}) : \mathsf{Dyn} \Rightarrow \mathsf{Int})$ evaluates to a *blame error*. This example shows that, for a value to be treated dynamically, it must be wrapped in a cast from its type to the dynamic type Dyn. Lets assume we would want to adapt the data type *MaybeInt* to be able to contain elements of any type:

$$\mathit{data}\ \mathit{MaybeDyn} = \mathit{Nothing} \mid \mathit{Just}\ \mathit{Dyn}$$

Using type constructors, we can represent *MaybeDyn* as Unit+Dyn. Then we can build expressions that interact with, and values of, type *MaybeDyn*:

$$\begin{aligned}
\mathit{Err}_1 &\triangleq \mathit{Error}\ \text{“}\mathit{Maybe.fromJust} : \mathit{Nothing}\text{”} \\
\mathit{isJust} &\triangleq \lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ n \Rightarrow \mathsf{false} \mid \mathsf{inr}\ v \Rightarrow \mathsf{true} \\
\mathit{fromJust} &\triangleq \lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ n \Rightarrow \mathit{Err}_1 \mid \mathsf{inr}\ v \Rightarrow v \\
\mathit{just4} &\triangleq \mathsf{inr}\ 4\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn} \\
\mathit{nothing} &\triangleq \mathsf{inl}\ \mathsf{unit}\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn} \\
\mathit{justTrue} &\triangleq \mathsf{inr}\ \mathsf{true}\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}
\end{aligned}$$

In this example, and in following examples, we will consider the error primitive *Error* which is used to propagate errors in the execution. The type system rule for *Error* is presented in an example below, and *Error* is treated semantically as a value. The flaw in this approach is when

[1] The expression is missing casts (those who cast a type to the same type) that are irrelevant to what we are trying to demonstrate.

these values go through cast insertion. According to the cast insertion rules *Inl* and *Inr* [9], no casts are inserted. The cast insertion produces the same value expressions ($just4$, *nothing* and $justTrue$) as above. Unlike the example above, where true was wrapped around a cast from it's type to dynamic, the elements contained in tags are not wrapped in casts. Therefore, they are not truly dynamically typed. The evaluation of the expression

$$fromJust\ justTrue\ +1$$

would halt at this point

$$\mathsf{true} : \mathsf{Dyn} \Rightarrow \mathsf{Int} : \mathsf{Int} \Rightarrow \mathsf{Int} : \mathsf{Int} \Rightarrow \mathsf{Int}$$
$$+$$
$$1 : \mathsf{Int} \Rightarrow \mathsf{Int}$$

instead of returning the expected *blame error*. This is due to the fact that $\mathsf{true} : \mathsf{Dyn} \Rightarrow \mathsf{Int}$ is not a value, and there is no cast that allows the reduction. Considering the previous example, what failed was that true had no cast. Therefore a possible solution is to insert that cast directly. Simulating dynamic elements in product or sum types can be accomplished with a beta expansion whose abstraction is annotated with the dynamic type. The abstraction annotated with the dynamic type will then insert the necessary cast.

$$justTrue \triangleq \mathsf{inr}\ ((\lambda x : \mathsf{Dyn}\ .\ x)\ \mathsf{true})\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}$$

This way, true will have cast $\mathsf{true} : \mathsf{Bool} \Rightarrow \mathsf{Dyn}$. Therefore, the evaluation would reduce to

$$\mathsf{true} : \mathsf{Bool} \Rightarrow \mathsf{Dyn} : \mathsf{Dyn} \Rightarrow \mathsf{Int} : \mathsf{Int} \Rightarrow \mathsf{Int} : \mathsf{Int} \Rightarrow \mathsf{Int}$$
$$+$$
$$1 : \mathsf{Int} \Rightarrow \mathsf{Int}$$

and that would result in a *blame error*. This approach also works with products, since we would only need to add the abstraction to dynamically type the elements of the pair. However, this approach is not very elegant, since the program would be flooded with beta expansions. There is also the question of blame tracking. The "blame" would be assigned to the beta expansions, rather than the algebraic data constructors.

Our goal is to extend the type system and cast insertion rules so that algebraic data type elements can be dynamically typed by adding explicit type annotations. The cast insertion algorithm then adds casts to ensure those elements are dynamically typed.

5.2 Dynamic Products and Sums

Algebraic data types are built from product and sum types. As such, in order to allow dynamic data types, product and sum types must also be allowed to contain dynamic elements. Here we extend the system in [9] to allow this.

Syntax

$$
\begin{array}{rll}
\textit{Expressions } e & ::= \ldots & \\
& \mid (e, e) \text{ as } T & \text{pair} \\
& \mid \mathsf{fst}\ e & \text{first projection} \\
& \mid \mathsf{snd}\ e & \text{second projection} \\
& \mid \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ \mathit{var} \Rightarrow e \mid \mathsf{inr}\ \mathit{var} \Rightarrow e & \text{case} \\
& \mid \mathsf{inl}\ e\ \mathsf{as}\ T & \text{tagging (left)} \\
& \mid \mathsf{inr}\ e\ \mathsf{as}\ T & \text{tagging (right)} \\
\textit{Types } T & ::= \ldots & \\
& \mid T \times T & \text{product type} \\
& \mid T + T & \text{sum type}
\end{array}
$$

$\boxed{\Gamma \vdash_G e : T}$ Typing

$$\frac{\Gamma \vdash_G e_1 : T_1' \qquad T_1 \sim T_1' \qquad \Gamma \vdash_G e_2 : T_2' \qquad T_2 \sim T_2'}{\Gamma \vdash_G (e_1, e_2)\ \mathsf{as}\ T_1 \times T_2 : T_1 \times T_2}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash_G e : PM \qquad PM \rhd T_1 \times T_2}{\Gamma \vdash_G \mathsf{fst}\ e : T_1}\ \textsc{First} \qquad \frac{\Gamma \vdash_G e : PM \qquad PM \rhd T_1 \times T_2}{\Gamma \vdash_G \mathsf{snd}\ e : T_2}\ \textsc{Last}$$

$$\frac{\begin{array}{c}\Gamma \vdash_G e : PM \qquad PM \rhd T_1' + T_2' \\ \Gamma, x_1 : T_1' \vdash_G e_1 : T_1 \qquad \Gamma, x_2 : T_2' \vdash_G e_2 : T_2 \\ T_1 \sqcup T_2 = T_J\end{array}}{\Gamma \vdash_G \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow e_1 \mid \mathsf{inr}\ x_2 \Rightarrow e_2\ : T_J}\ \textsc{Case} \qquad \frac{\Gamma \vdash_G e : T_1' \qquad T_1 \sim T_1'}{\Gamma \vdash_G \mathsf{inl}\ e\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2}\ \textsc{Inl}$$

$$\frac{\Gamma \vdash_G e : T_2' \qquad T_2 \sim T_2'}{\Gamma \vdash_G \mathsf{inr}\ e\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2}\ \textsc{Inr}$$

Figure 5.1: Gradual Type System (Products and Sums)

5.2.1 Syntax

We must provide a way to explicitly annotate values of sum and product types (pairs and the tags inl and inr). These type annotations will later be used to derive the type of the expression. As inl and inr already have explicit type annotations (necessary to ensure uniqueness of types) in [9], only the remaining value, pair, has to be altered to allow a type annotation. Figure 5.1 presents the syntax for these terms. The new syntax allows to specify which elements are to be dynamically typed by inserting the appropriate type annotation in the value expression.

5.2.2 Type System

We must also make extensions to the type system described in [9], so that the value is typed with the type in the annotation. These extensions enable the term, inside a pair or a tag, to be typed with the dynamic type. Figure 5.1 shows the extended type system.

Only the rules *Pair*, *Inl* and *Inr* differ from the (standard) gradual type system [9], because these are the rules that type values. In the case of the rule *Pair*, we now type the pair with the type present in the annotation. However, we must still make sure that the type in the annotation, and therefore the final type of the pair, is consistent with the elements being contained in the pair. Therefore, we need the consistency relations $T_1 \sim T_1'$ and $T_2 \sim T_2'$ to ensure the types are consistent. By having consistency ($\sim$) instead of equality ($=$) relations, we allow the dynamic type to be inserted in the pair's type annotation. The same goes for the rules *Inl* and *Inr*.

With these extensions, the values contained in a pair or in a tag can be typed according to the type annotation. Therefore, we can type these values with the dynamic type. For example, consider the following expression:

$$just4 \triangleq \mathsf{inr}\ 4\ \mathsf{as}\ \mathsf{Unit} + \mathsf{Dyn}$$

Without these extensions, this term could not be expressed simply because the dynamic type could not be added to a type annotation. However, with the extensions to the syntax and type system, this term is now valid and has type $\mathsf{Unit} + \mathsf{Dyn}$.

5.2.3 Cast Insertion

Finally, the cast insertion [9] rules must be extended to allow the insertion of key casts that will allow the expression to be evaluated in a correct manner. Figure 5.2 presents the cast insertion rules.

These rules must ensure the necessary casts are inserted in the sub-terms of pairs, inl and inr. The rule *Pair* inserts casts in both sub-terms of pair. For each sub-term, a cast from the type of the sub-term to the type specified in the annotation is inserted. The rules *Inl* and *Inr* are similar, and the only difference is what type is chosen as the target type of the cast, according to the tag. Although the main reason for these casts to be inserted is to enable values to be dynamically typed, these same casts are inserted even if the type annotations are not dynamic. If the type annotation is not consistent with the type of the sub-term, the type system should have caught the type error earlier. If the type annotation is equal to the type of the sub-term, we then end up with a cast from a type to the same type, that will be eliminated during evaluation using the reduction rule ID-BASE from [10].

$\boxed{\Gamma \vdash_{CC} e \rightsquigarrow e' : T}$ Cast Insertion

$$\frac{\Gamma \vdash_{CC} e_1 \rightsquigarrow e_1' : T_1' \qquad \Gamma \vdash_{CC} e_2 \rightsquigarrow e_2' : T_2'}{\Gamma \vdash_{CC} (e_1, e_2) \text{ as } T_1 \times T_2 \rightsquigarrow (e_1' : T_1' \Rightarrow T_1, e_2' : T_2' \Rightarrow T_2) \text{ as } T_1 \times T_2 : T_1 \times T_2} \textsc{Pair}$$

$$\frac{\Gamma \vdash_{CC} e \rightsquigarrow e' : PM \qquad PM \rhd T_1 \times T_2}{\Gamma \vdash_{CC} \mathsf{fst}\ e \rightsquigarrow \mathsf{fst}\ (e' : PM \Rightarrow T_1 \times T_2) : T_1} \textsc{First}$$

$$\frac{\Gamma \vdash_{CC} e \rightsquigarrow e' : PM \qquad PM \rhd T_1 \times T_2}{\Gamma \vdash_{CC} \mathsf{snd}\ e \rightsquigarrow \mathsf{snd}\ (e' : PM \Rightarrow T_1 \times T_2) : T_2} \textsc{Last}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{CC} e \rightsquigarrow e' : PM \qquad PM \rhd T_1' + T_2' \\ \Gamma, x : T_1' \vdash_{CC} e_1 \rightsquigarrow e_1' : T_1 \qquad \Gamma, x : T_2' \vdash_{CC} e_2 \rightsquigarrow e_2' : T_2 \qquad T_1 \sqcup T_2 = T_J\end{array}}{\begin{array}{l}\Gamma \vdash_{CC} \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow e_1 \mid \mathsf{inr}\ x_2 \Rightarrow e_2 \ \rightsquigarrow \\ \mathsf{case}\ (e' : PM \Rightarrow T_1' + T_2')\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow (e_1' : T_1 \Rightarrow T_J) \mid \mathsf{inr}\ x_2 \Rightarrow (e_2' : T_2 \Rightarrow T_J)\ : T_J\end{array}} \textsc{Case}$$

$$\frac{\Gamma \vdash_{CC} e \rightsquigarrow e' : T_1'}{\Gamma \vdash_{CC} \mathsf{inl}\ e\ \mathsf{as}\ T_1 + T_2 \rightsquigarrow \mathsf{inl}\ (e' : T_1' \Rightarrow T_1)\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2} \textsc{Inl}$$

$$\frac{\Gamma \vdash_{CC} e \rightsquigarrow e' : T_2'}{\Gamma \vdash_{CC} \mathsf{inr}\ e\ \mathsf{as}\ T_1 + T_2 \rightsquigarrow \mathsf{inr}\ (e' : T_2' \Rightarrow T_2)\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2} \textsc{Inr}$$

Figure 5.2: Cast Insertion (Products and Sums)

5.3 Dynamic Tuples and Variants

Here we generalize the previous system to deal with tuples and variants. Tuples may have an arbitrary number of elements. In the case of tuple projection, we define a projection primitive for any size of a tuple, and the projection requires the size of the tuple. The syntax π_i^n e (sometimes also written as $\mathsf{proj}\ i\ n\ e$) means that we are projecting from a expression e, and that the projection is expecting a tuple of size n, and we want to project the i^{th} element of that tuple. Similarly, the term case may now have an arbitrary number of alternatives.

Then we allow the description of structures of arbitrary size. $(e_i\ ^{i \in 1..n})$ for a tuple with n terms and $(T_i\ ^{i \in 1..n})$ for it's type. $\langle l_i : T_i\ ^{i \in 1..n} \rangle$ stands for the variant type with n alternatives, where e_i are expressions, l_i are labels and T_i are types.

Syntax

$$
\begin{array}{rll}
\textit{Expressions } e & ::= \ldots & \\
& \mid (e_i\ ^{i\in 1..n})\ \textit{as}\ T & \text{tuple} \\
& \mid \pi_n^n\ e & \text{projection} \\
& \mid \mathsf{case}\ e\ \mathsf{of}\ \langle l_i = x_i\rangle \Rightarrow e_i\ ^{i\in 1..n} & \text{case} \\
& \mid \langle l = e\rangle\ \mathsf{as}\ T & \text{tagging} \\
\textit{Types } T & ::= \ldots & \\
& \mid (T_i\ ^{i\in 1..n}) & \text{tuple type} \\
& \mid \langle l_i : T_i\ ^{i\in 1..n}\rangle & \text{variant type}
\end{array}
$$

$\boxed{\Gamma \vdash_G e : T}$ Typing

$$\frac{\text{for each } i \qquad \Gamma \vdash_G e_i : T_i' \qquad T_i \sim T_i'}{\Gamma \vdash_G (e_i\ ^{i\in 1..n})\ \mathsf{as}\ (T_i\ ^{i\in 1..n}) : (T_i\ ^{i\in 1..n})}\ \textsc{Tuple}$$

$$\frac{\Gamma \vdash_G e : PM \qquad PM \rhd (T_j\ ^{j\in 1..n})}{\Gamma \vdash_G \pi_i^n\ e : T_i}\ \textsc{Projection}$$

$$\frac{\begin{array}{c}\Gamma \vdash_G e : PM \qquad PM \rhd \langle l_i : T_i'\ ^{i\in 1..n}\rangle \\ \text{for each } i \qquad \Gamma, x_i : T_i' \vdash_G e_i : T_i \\ T_1 \sqcup \ldots \sqcup T_n = T_J\end{array}}{\Gamma \vdash_G \mathsf{case}\ e\ \mathsf{of}\ \langle l_i = x_i\rangle \Rightarrow e_i\ ^{i\in 1..n} : T_J}\ \textsc{Case}$$

$$\frac{\Gamma \vdash_G e_j : T_j' \qquad T_j \sim T_j'}{\Gamma \vdash_G \langle l_j = e_j\rangle\ \mathsf{as}\ \langle l_i : T_i\ ^{i\in 1..n}\rangle : \langle l_i : T_i\ ^{i\in 1..n}\rangle}\ \textsc{Tag}$$

Figure 5.3: Gradual Type System (Tuples and Variants)

5.3.1 Syntax

An explicit type annotation is needed in a tuple (as was needed in a pair) and tag, so that the types of the elements can be specified. In the case of tag, there is already a type annotation, so only the tuple term changes. Figure 5.3 presents the syntax.

5.3.2 Type System

As with products and sums, extensions are necessary to the type system of tuples and variants. These are similar to those in Figure 5.1, however, we must take into account that these generalizations do not have a fixed structure, but have an arbitrary number of terms. The type

system is presented in Figure 5.3.

Some subtleties that are present in this type system are worth mentioning. In the case of tuple projection, we need to indicate the size of the tuple we expect to project from, as stated previously. This is a necessary approach, because when we pattern match ($\triangleright$) the type PM (in the *Projection* rule), we must ensure it has the expected size. If PM is the dynamic type, then we must be able to construct the tuple type of size n. The information about the size of the tuple is necessary to guide the gradual type system. If we were to define a different projection term for each different size that the tuple can have (*fst3*, *fst4*, *fst5*, ..., possibly up to an upper limit), then this information would be implicitly present, because each term would be supposed to project from a tuple of a certain size. This is also the case in the rule *Case*, but with slight differences. While in the *Projection* rule, we must ensure PM has the expected size, in the *Case* rule we must ensure that PM has the same labels as the labels present in the alternatives.

To exemplify the changes proposed so far, let's take a look at how a maybe containing a dynamic element can be built. We start with the data type definition:

$$data\ MaybeDyn = Nothing \mid Just\ Dyn$$

This means the type *MaybeDyn* can either contain nothing or a element whose type is unknown. The representation of *MaybeDyn* using type constructors is:

$$\langle Nothing : Unit, Just : Dyn \rangle$$

Now that we have the correct type, we can build expressions that interact with, and values of, type *MaybeDyn*.

```
let isJust = \maybe . case maybe of
    <Nothing=x> => false
  | <Just=v> => true in
let fromJust = \maybe . case maybe of
    <Nothing=x> => Error "Maybe.fromJust: Nothing"
  | <Just=v> => v in
let just4 = <Just=4> as <Nothing:Unit, Just:Dyn> in
let nothing = <Nothing=unit> as <Nothing:Unit, Just:Dyn> in
let justTrue = <Just=True> as <Nothing:Unit, Just:Dyn> in
fromJust justTrue + 1
```

The values that the variables *just*4, *nothing* and *justTrue* bind have type *MaybeDyn* due to the type annotations. Note that the type annotation in values is what decides the type of the value. To demonstrate how the changed type system rules work, consider the typing derivation of the expression defined above. For the sake of readability and to keep the presentation of the

typing derivation small, we will ignore unused expressions and use the following abbreviations:

$$
\begin{aligned}
MaybeDyn &\triangleq \langle Nothing : \mathsf{Unit}, Just : \mathsf{Dyn} \rangle \\
Err_2 &\triangleq Error\ \text{“}Maybe.fromJust : Nothing\text{”} \\
fromJust &\triangleq \lambda m\ .\ \mathsf{case}\ m\ \mathsf{of}\ \langle Nothing = x \rangle \Rightarrow Err_2 \mid \langle Just = v \rangle \Rightarrow v \\
justTrue &\triangleq \langle Just = \mathsf{true} \rangle\ \mathsf{as}\ MaybeDyn
\end{aligned}
$$

To type this expression, an extra typing rule is necessary:

$$\frac{}{\vdash_G Error\ var : T}\ \textsc{Error}$$

$$\frac{}{m : MaybeDyn \vdash_G m : MaybeDyn}\ \textsc{Var} \qquad MaybeDyn \rhd MaybeDyn$$

$$\frac{}{m : MaybeDyn, x : \mathsf{Unit} \vdash_G Err_2 : \mathsf{Dyn}}\ \textsc{Error}$$

$$\frac{}{m : MaybeDyn, v : \mathsf{Dyn} \vdash_G v : \mathsf{Dyn}}\ \textsc{Var} \qquad \mathsf{Dyn} \sqcup \mathsf{Dyn} = \mathsf{Dyn}$$

$$2\ \frac{\dfrac{}{m : MaybeDyn \vdash_G \mathsf{case}\ m\ \mathsf{of}\ \langle Nothing = x \rangle \Rightarrow Err_2 \mid \langle Just = v \rangle \Rightarrow v : \mathsf{Dyn}}\ \textsc{Case}}{\vdash_G \lambda m\ .\ \mathsf{case}\ m\ \mathsf{of}\ \langle Nothing = x \rangle \Rightarrow Err_2 \mid \langle Just = v \rangle \Rightarrow v : MaybeDyn \to \mathsf{Dyn}}\ \textsc{Abs}$$

$$3\ \frac{\dfrac{}{\vdash_G \mathsf{true} : \mathsf{Bool}}\ \textsc{True} \qquad \mathsf{Dyn} \sim \mathsf{Bool}}{\vdash_G \langle Just = \mathsf{true} \rangle\ \mathsf{as}\ MaybeDyn : MaybeDyn}\ \textsc{Tag}$$

$$1\ \frac{2\ \dfrac{}{\vdash_G fromJust : MaybeDyn \to \mathsf{Dyn}}\ \textsc{Abs} \qquad MaybeDyn \to \mathsf{Dyn} \rhd MaybeDyn \to \mathsf{Dyn} \qquad 3\ \dfrac{}{\vdash_G justTrue : MaybeDyn}\ \textsc{Tag} \qquad MaybeDyn \sim MaybeDyn}{\vdash_G fromJust\ justTrue : \mathsf{Dyn}}\ \textsc{App}$$

$$\frac{1\ \dfrac{}{\vdash_G fromJust\ justTrue : \mathsf{Dyn}}\ \textsc{App} \qquad \mathsf{Dyn} \rhd \mathsf{Int} \qquad \dfrac{}{\vdash_G 1 : \mathsf{Int}}\ \textsc{Int} \qquad \mathsf{Int} \rhd \mathsf{Int}}{\vdash_G fromJust\ justTrue + 1 : \mathsf{Int}}\ \textsc{Add}$$

The expression is typed with type Int although it contains a type error, which will only be discovered during runtime.

$\boxed{\Gamma \vdash_{CC} e \rightsquigarrow e' : T}$ Cast Insertion

$$\frac{\text{for each } i \qquad \Gamma \vdash_{CC} e_i \rightsquigarrow e_i' : T_i'}{\Gamma \vdash_{CC} (e_i\ ^{i\in 1..n})\ \mathsf{as}\ (T_i\ ^{i\in 1..n}) \rightsquigarrow (e_i : T_i' \Rightarrow T_i\ ^{i\in 1..n})\ \mathsf{as}\ (T_i\ ^{i\in 1..n}) : (T_i\ ^{i\in 1..n})}\ \textsc{Tuple}$$

$$\frac{\Gamma \vdash_{CC} e \rightsquigarrow e' : PM \qquad PM \rhd (T_j\ ^{j\in 1..n})}{\Gamma \vdash_{CC} \pi_i^n\ e \rightsquigarrow \pi_i^n\ (e' : PM \Rightarrow (T_j\ ^{j\in 1..n})) : T_i}\ \textsc{Projection}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{CC} e \rightsquigarrow e' : PM \qquad PM \rhd \langle l_i : T_i'\ ^{i\in 1..n}\rangle \\ \text{for each } i \qquad \Gamma, x_i : T_i' \vdash_G e_i \rightsquigarrow e_i' : T_i \qquad T_1 \sqcup ... \sqcup T_n = T_J\end{array}}{\begin{array}{l}\Gamma \vdash_{CC} \mathsf{case}\ e\ \mathsf{of}\ \langle l_i = x_i\rangle \Rightarrow e_i\ ^{i\in 1..n} \rightsquigarrow \\ \mathsf{case}\ (e' : PM \Rightarrow \langle l_i : T_i'\ ^{i\in 1..n}\rangle)\ \mathsf{of}\ \langle l_i = x_i\rangle \Rightarrow (e_i : T_i \Rightarrow T_J)\ ^{i\in 1..n} : T_J\end{array}}\ \textsc{Case}$$

$$\frac{\Gamma \vdash_{CC} e_j \rightsquigarrow e_j' : T_j'}{\Gamma \vdash_{CC} \langle l_j = e_j\rangle\ \mathsf{as}\ \langle l_i : T_i\ ^{i\in 1..n}\rangle \rightsquigarrow \langle l_j = (e_j : T_j' \Rightarrow T_j)\rangle\ \mathsf{as}\ \langle l_i : T_i\ ^{i\in 1..n}\rangle : \langle l_i : T_i\ ^{i\in 1..n}\rangle}\ \textsc{Tag}$$

Figure 5.4: Cast Insertion (Tuples and Variants)

5.3.3 Cast Insertion

The cast insertion rules for tuples (*Tuple*) and variants (*Tag*) are similar to those for products (*Pair*) and sums (*Inl* and *Inr*). The rules are presented in Figure 5.4. In the rule *Tuple*, instead of adding casts to the two elements of pair, we now must add casts for each element of the tuple. The rule *Tag* is similar to the rules *Inl* and *Inr*. The difference is that instead of choosing the left or right side, the rule must obtain the type, that will be the cast destination, from the type annotation by comparing the label in the tag expression with the labels in the type annotation.

Considering the previous example, when the casts are inserted in the expression and it is evaluated, the execution does not halt, like it does with the analogous example (with sum types) at the start of Chapter 5, but results in a *blame error*. By comparing the execution of this example

```
((((True : Bool => Dyn) : Dyn => Int): Int => Int)
: Int => Int) + (1 : Int => Int)
```

with the execution of the example at the start of Chapter 5

```
(((True : Dyn => Int) : Int => Int)
: Int => Int) + (1 : Int => Int)
```

we can see that, with the proposed cast insertion rules, the necessary cast is present in the expression. Therefore, the cast

```
((True : Bool => Dyn) : Dyn => Int)
```

reduces to a *blame error*, which is then pushed to the top level and returned as the result of the evaluation.

The expressions $isJust$ and $fromJust$ may also be applied to terms of type *MaybeInt* ($\langle Nothing : \mathsf{Unit}, Just : \mathsf{Int}\rangle$). The resulting expression will then be statically typed and evaluated thereafter. In order to choose between static and dynamic typing, we must now insert the appropriate type in the type annotation in the tag.

5.4 Recursive Types

The type constructors referred so far are the building blocks for the creation of new types. However, there is another building block for the creation of recursive data types. The type constructor μ [23] is the type constructor that allows recursive type definitions. Although the treatment of this type constructor does not require any extension in order to allow dynamic data types, in accordance with the scope of this dissertation, we thought it would be helpful to describe the type system and cast insertion rules. Although we chose to represent the iso-recursive approach to recursive types, we could have chosen the equi-recursive approach [23]. However, the gradual type system for iso-recursive approach is already defined [9] and it is easier to reason with.

5.4.1 Syntax

Although no changes to the syntax of recursive types are required, Figure 5.5 presents the syntax.

5.4.2 Type System

The standard terms for the iso-recursive type are fold and unfold. These terms are used to guide the type system in type checking expressions with recursive types. The type system rules are similar to those of the static type system (such as in [23]), however they have some differences. In the *Unfold* rule, a pattern matching relation is needed, because we expect the term e to be typed with a recursive type. The term e may be typed with the dynamic type, and if that's the case, we must convert the dynamic type to the recursive type. Furthermore, we require the consistency relation between the type of the sub-term e and the expected type for the sub-term, which in *Fold* is a one-step unfolding of the type in the annotation in fold and in *Unfold* is the recursive type present in the annotation of unfold. The type rules are shown in Figure 5.5.

To show how recursive types can be used to define recursive gradual data types, let's now

Syntax

$$
\begin{array}{rll}
\textit{Expressions } e & ::= & \ldots \\
& \mid \mathsf{fold}\ [T]\ e & \text{folding} \\
& \mid \mathsf{unfold}\ [T]\ e & \text{unfolding} \\
\textit{Types } T & ::= & \ldots \\
& \mid \mu X\ .\ T & \text{recursive type}
\end{array}
$$

$\boxed{\Gamma \vdash_G e : T}$ Typing

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash_G e : T' \qquad [X \mapsto U]T \ \sim\ T'}{\Gamma \vdash_G \mathsf{fold}\ [U]\ e : U}\ \textsc{Fold}$$

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash_G e : PM \qquad PM \triangleright \mu X'\ .\ T' \qquad U\ \sim\ \mu X'\ .\ T'}{\Gamma \vdash_G \mathsf{unfold}\ [U]\ e : [X \mapsto U]T}\ \textsc{Unfold}$$

Figure 5.5: Gradual Type System (Recursive types)

consider building lists whose elements are of unknown type. The data type

$$data\ ListDyn = Nil \mid Cons\ Dyn\ ListDyn$$

can be used to define lists as a recursive type [23] using a equi-recursive approach. However, we take the iso-recursive approach and define lists as:

$$\mu L.\langle Nil : Unit, Cons : (Dyn, L)\rangle$$

We will be using the fold and unfold terms (to enable recursive types). Since fold and unfold, like values, require type annotations, for the sake of readability, we will use the names *ListDyn* and *ListDyn'* as abbreviations of the *ListDyn* type and it's one-step unfolding, respectively:

```
ListDyn  = rec l . <Nil:Unit, Cons:(Dyn,l)>
ListDyn' = <Nil:Unit, Cons:(Dyn,rec l . <Nil:Unit, Cons:(Dyn,l)>)>
```

The standard terms for building generic lists as well as the lists themselves can be defined in the following manner:

```
let nil = fold [ListDyn] <Nil=unit> as ListDyn' in
let cons = \v . \l . fold [ListDyn]
  <Cons=(v, l) as (Dyn, ListDyn)> as ListDyn' in
let isnil = \l . case (unfold [ListDyn] l) of
    <Nil=n> => True
```

```
  | <Cons=c> => False in
let hd = \l . case (unfold [ListDyn] l) of
    <Nil=n> => Error "List.head: empty list"
  | <Cons=c> => proj 1 2 c in
let tl = \l . case (unfold [ListDyn] l) of
    <Nil=n> => Error "List.tail: empty list"
  | <Cons=c> => proj 2 2 c in
let list1 = cons true nil in
hd list1 + 1
```

To demonstrate how the changed type system rules work, consider the typing derivation of the expression defined above. For the sake of readability and to keep the presentation of the typing derivation small, we will ignore unused expressions and use the following abbreviations:

$$
\begin{aligned}
ListDyn &\triangleq \mu L.\langle Nil : Unit, Cons : (Dyn, L)\rangle \\
ListDyn' &\triangleq \langle Nil : Unit, Cons : (Dyn, ListDyn)\rangle \\
Err_3 &\triangleq Error\ \text{“}List.head : empty\ list\text{”} \\
nil &\triangleq \mathsf{fold}\ [ListDyn]\ \langle Nil = \mathsf{unit}\rangle\ \mathsf{as}\ ListDyn' \\
cons &\triangleq \lambda v\ .\ \lambda l\ .\ \mathsf{fold}\ [ListDyn]\ \langle Cons = (v,\ l)\ \mathsf{as}\ (\mathsf{Dyn},\ ListDyn)\rangle\ \mathsf{as}\ ListDyn' \\
hd &\triangleq \lambda l\ .\ \mathsf{case}\ (\mathsf{unfold}\ [ListDyn]\ l)\ \mathsf{of}\ \langle Nil = n\rangle \Rightarrow Err_3 \mid \langle Cons = c\rangle \Rightarrow \pi_1^2\ c \\
list_1 &\triangleq \mathsf{cons\ true\ nil}
\end{aligned}
$$

To type this expression, an extra typing rule is necessary:

$$\frac{}{\vdash_G Error\ var : T}\ \textsc{Error}$$

$$3\ \frac{\dfrac{}{l : ListDyn \vdash_G l : ListDyn}\ \textsc{Var} \qquad ListDyn \rhd ListDyn \qquad ListDyn \sim ListDyn}{l : ListDyn \vdash_G \mathsf{unfold}\ [ListDyn]\ l : ListDyn'}\ \textsc{Unfold}$$

$$4\ \frac{\dfrac{}{l : ListDyn, c : (\mathsf{Dyn}, ListDyn) \vdash_G c : (\mathsf{Dyn}, ListDyn)}\ \textsc{Var} \qquad (\mathsf{Dyn}, ListDyn) \rhd (\mathsf{Dyn}, ListDyn)}{l : ListDyn, c : (\mathsf{Dyn}, ListDyn) \vdash_G \pi_1^2\ c : \mathsf{Dyn}}\ \textsc{Projection}$$

$$\dfrac{3\ \dfrac{}{l : ListDyn \vdash_G \mathsf{unfold}\ [ListDyn]\ l : ListDyn'}\ \textsc{Unfold}}{\dfrac{}{l : ListDyn, n : \mathsf{Unit} \vdash_G Err_3 : \mathsf{Dyn}}\ \textsc{Error}}$$

$$ListDyn' \rhd ListDyn'$$

$$\dfrac{4\ \dfrac{}{l : ListDyn, c : (\mathsf{Dyn}, ListDyn) \vdash_G \pi_1^2\ c : \mathsf{Dyn}}\ \textsc{Projection}}{\dfrac{\dfrac{\mathsf{Dyn} \sqcup \mathsf{Dyn} = \mathsf{Dyn}}{l : ListDyn \vdash_G \mathsf{case}\ (\mathsf{unfold}\ [ListDyn]\ l)\ \mathsf{of}\ \langle Nil = n\rangle \Rightarrow Err_3 \mid \langle Cons = c\rangle \Rightarrow \pi_1^2\ c : \mathsf{Dyn}}\ \textsc{Case}}{2\ \dfrac{}{\vdash_G \lambda l\ .\ \mathsf{case}\ (\mathsf{unfold}\ [ListDyn]\ l)\ \mathsf{of}\ \langle Nil = n\rangle \Rightarrow Err_3 \mid \langle Cons = c\rangle \Rightarrow \pi_1^2\ c : ListDyn \to \mathsf{Dyn}}\ \textsc{Abs}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{v : \mathsf{Dyn}}\ \textsc{Var} \quad \dfrac{}{l : ListDyn}\ \textsc{Var}}{\mathsf{Dyn} \sim \mathsf{Dyn} \quad ListDyn \sim ListDyn}}{v : \mathsf{Dyn}, l : ListDyn \vdash_G (v,\ l)\ \mathsf{as}\ (\mathsf{Dyn},\ ListDyn) : (\mathsf{Dyn}, ListDyn)}\ \textsc{Tuple}}{(\mathsf{Dyn}, ListDyn) \sim (\mathsf{Dyn}, ListDyn)}}{v : \mathsf{Dyn}, l : ListDyn \vdash_G \langle Cons = (v,\ l)\ \mathsf{as}\ (\mathsf{Dyn},\ ListDyn)\rangle\ \mathsf{as}\ ListDyn' : ListDyn'}\ \textsc{Tag}}{ListDyn' \sim ListDyn'}}{8\ \dfrac{}{v : \mathsf{Dyn}, l : ListDyn \vdash_G \mathsf{fold}\ [ListDyn]\ \langle Cons = (v,\ l)\ \mathsf{as}\ (\mathsf{Dyn},\ ListDyn)\rangle\ \mathsf{as}\ ListDyn' : ListDyn}\ \textsc{Fold}}$$

$$\dfrac{\dfrac{8\ \dfrac{}{v : \mathsf{Dyn}, l : ListDyn \vdash_G \mathsf{fold}\ [ListDyn]}\ \textsc{Fold}}{\langle Cons = (v,\ l)\ \mathsf{as}\ (\mathsf{Dyn},\ ListDyn)\rangle\ \mathsf{as}\ ListDyn' : ListDyn}}{v : \mathsf{Dyn} \vdash_G \lambda l\ .\ \mathsf{fold}\ [ListDyn]}\ \textsc{Abs}$$

$$7\ \dfrac{\langle Cons = (v,\ l)\ \mathsf{as}\ (\mathsf{Dyn},\ ListDyn)\rangle\ \mathsf{as}\ ListDyn' : ListDyn \to ListDyn}{\vdash_G \lambda v\ .\ \lambda l\ .\ \mathsf{fold}\ [ListDyn]\ \langle Cons = (v,\ l)\ \mathsf{as}\ (\mathsf{Dyn},\ ListDyn)\rangle\ \mathsf{as}\ ListDyn' : \mathsf{Dyn} \to (ListDyn \to ListDyn)}\ \textsc{Abs}$$

$$6\ \dfrac{\dfrac{7\ \dfrac{}{\vdash_G \mathsf{cons} : \mathsf{Dyn} \to (ListDyn \to ListDyn)}\ \textsc{Abs} \quad \dfrac{}{\vdash_G \mathsf{true} : \mathsf{Bool}}\ \textsc{True}}{\mathsf{Dyn} \to (ListDyn \to ListDyn) \rhd \mathsf{Dyn} \to (ListDyn \to ListDyn) \quad \mathsf{Bool} \sim \mathsf{Dyn}}}{\vdash_G \mathsf{cons\ true}\ : ListDyn \to ListDyn}\ \textsc{App}$$

$$9\ \dfrac{\dfrac{}{\vdash_g \mathsf{unit} : \mathsf{Unit}}\ \textsc{Unit} \qquad \mathsf{Unit} \sim \mathsf{Unit}}{\vdash_G \langle Nil = \mathsf{unit}\rangle\ \mathsf{as}\ ListDyn' : ListDyn'}\ \textsc{Tag} \qquad ListDyn' \sim ListDyn'}{\vdash_G \mathsf{fold}\ [ListDyn]\ \langle Nil = \mathsf{unit}\rangle\ \mathsf{as}\ ListDyn' : ListDyn}\ \textsc{Fold}$$

$$5\ \dfrac{6\ \dfrac{}{\vdash_G \mathsf{cons\ true} : ListDyn \to ListDyn}\ \textsc{App} \qquad 9\ \dfrac{}{\vdash_G \mathsf{nil} : ListDyn}\ \textsc{Fold}}{\vdash_G \mathsf{cons\ true\ nil} : ListDyn}\ \textsc{App}$$

$$1\ \dfrac{2\ \dfrac{}{\vdash_G hd : ListDyn \to \mathsf{Dyn}}\ \textsc{Abs} \qquad 5\ \dfrac{}{\vdash_G list_1 : ListDyn}\ \textsc{Abs}}{\vdash_G hd\ list_1 : \mathsf{Dyn}}\ \textsc{App}$$

$$\dfrac{1\ \dfrac{}{\vdash_G hd\ list_1 : \mathsf{Dyn}}\ \textsc{App} \qquad \mathsf{Dyn} \rhd \mathsf{Int} \qquad \dfrac{}{\vdash_G 1 : \mathsf{Int}}\ \textsc{Int} \qquad \mathsf{Int} \rhd \mathsf{Int}}{\vdash_G hd\ list_1\ + 1 : \mathsf{Int}}\ \textsc{Add}$$

The expression is typed with type Int although it contains a type error, which will only be discovered during runtime.

5.4.3 Cast Insertion

The cast insertion rules are a relatively straightforward transformation from the type system rules. In the *Fold* rule, no cast is inserted while in the *Unfold* rule only the cast pertaining the pattern matching relation is inserted. The cast insertion rules for the recursive type are shown in Figure 5.6.

5.5 Type Inference

Type inference for gradual typing was previously defined in [14], where type inference is extended to a solver which deals with consistency constraints. In this section we extend the type inference algorithm to deal with dynamic algebraic data types (products, sums, tuples and variants).

These rules are an extension to the rules in [14], and as such follow the same conventions.

$\boxed{\Gamma \vdash_{CC} e \rightsquigarrow e' : T}$ Cast Insertion

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash_{CC} e \rightsquigarrow e' : T' \qquad [X \mapsto U]\ T \stackrel{\sim}{\sim} T'}{\Gamma \vdash_{CC} \mathsf{fold}\ [U]\ e \rightsquigarrow \mathsf{fold}\ [U]\ e' : U}\ \textsc{Fold}$$

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash_{CC} e \rightsquigarrow e' : PM \qquad PM \rhd \mu X'\ .\ T' \qquad U \stackrel{\sim}{\sim} \mu X'\ .\ T'}{\Gamma \vdash_{CC} \mathsf{unfold}\ [U]\ e \rightsquigarrow \mathsf{unfold}\ [U]\ (e' : PM \Rightarrow \mu X'\ .\ T') : [X \mapsto U]T}\ \textsc{Unfold}$$

Figure 5.6: Cast Insertion (Recursive types)

The constraint generation judgement $\Gamma \vdash e : T \mid_\chi C$ means that given a context Γ and an expression e, e can be given type T if the constraints C can be satisfied, and χ denotes the variables used to express the constraints. The symbols $\stackrel{\sim}{\sim}$ and $\doteq$ are used to represent the consistency and the equality constraints, respectively. Constraint generation rules for products and sums are presented in Figure 5.7, for tuples and variants in Figure 5.9 and for the recursive type in Figure 5.11. The constraint solving judgement $C\ \upsilon\ S$ means that the constraints C are solved, producing a set of substitutions S. The constraint solving rules are relatively simple, since these rules share a fixed structure with the rules in [14]. For each type constructor, there are 3 different rules that must be derived: two rules for solving equality and consistency constraints between two types with the same type constructor and a rule for solving a consistency constraint between a type variable and a type. Constraint solving rules for products and sums are present in Figure 5.8, for tuples and variants in Figure 5.10 and for the recursive type in Figure 5.12. There are some aspects that are not so straightforward and thus require additional explanation.

The meet relation between types ($\sqcap$) is defined in [14], and although the rules for product, tuple, sum and variant types are not presented in that paper, they can be derived in a straightforward manner. The meet relation is similar to the join relation ($\sqcup$), that is defined in [9] because both return the type that is the least upper bound w.r.t. the precision relation.

Pattern matching ($\rhd$) is paramount in order to convert a dynamic type to it's expected type. Types whose structures are of arbitrary size, complicate pattern matching. For example, consider the following pattern matching judgement:

$$PM \rhd T_1 + T_2 \mid_\chi C$$

This means that we will pattern match the type PM to some type whose structure is that of a sum ($+$) type. The pattern matching relation in a sense takes the input type and a type constructor and returns a type or fails. However, when dealing with types constructors of arbitrary size, things complicate. Consider the pattern matching judgement for tuple type:

$$PM \rhd (T_i\ ^{i \in 1..n}) \mid_\chi C$$

$\boxed{\Gamma \vdash e : T \mid_\chi C}$ Constraint Generation

$$\frac{\Gamma \vdash e_1 : T_1' \mid_{\chi_1} C_1 \qquad \Gamma \vdash e_2 : T_2' \mid_{\chi_2} C_2}{\Gamma \vdash (e_1, e_2)\ \mathsf{as}\ T_1 \times T_2 : T_1 \times T_2 \mid_{\chi_1 \cup \chi_2} C_1 \cup C_2 \cup \{T_1 \dot{\sim} T_1', T_2 \dot{\sim} T_2'\}}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash e : PM \mid_\chi C \qquad PM \rhd T_1 \times T_2 \mid_{\chi_1} C_1}{\Gamma \vdash \mathsf{fst}\ e : T_1 \mid_{\chi \cup \chi_1} C \cup C_1}\ \textsc{First}$$

$$\frac{\Gamma \vdash e : PM \mid_\chi C \qquad PM \rhd T_1 \times T_2 \mid_{\chi_1} C_1}{\Gamma \vdash \mathsf{snd}\ e : T_2 \mid_{\chi \cup \chi_1} C \cup C_1}\ \textsc{Second}$$

$$\frac{\begin{array}{c}\Gamma \vdash\ e : PM \mid_\chi C \qquad PM \rhd T_1' + T_2' \mid_{\chi'} C' \\ \Gamma, x_1 : T_1' \vdash e_1 : T_1 \mid_{\chi_1} C_1 \qquad \Gamma, x_2 : T_2' \vdash e_2 : T_2 \mid_{\chi_2} C_2 \qquad T_1 \sqcap T_2 = T_J \mid_{\chi_3} C_3\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x_1 \Rightarrow e_1 \mid \mathsf{inr}\ x_2 \Rightarrow e_2\ : T_J \mid_{\chi \cup \chi' \cup \chi_1 \cup \chi_2 \cup \chi_3} C \cup C' \cup C_1 \cup C_2 \cup C_3}\ \textsc{Case}$$

$$\frac{\Gamma \vdash e : T_1' \mid_\chi C}{\Gamma \vdash \mathsf{inl}\ e\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2 \mid_\chi C \cup \{T_1 \dot{\sim} T_1'\}}\ \textsc{Inl}$$

$$\frac{\Gamma \vdash e : T_2' \mid_\chi C}{\Gamma \vdash \mathsf{inr}\ e\ \mathsf{as}\ T_1 + T_2 : T_1 + T_2 \mid_\chi C \cup \{T_2 \dot{\sim} T_2'\}}\ \textsc{Inr}$$

$\boxed{PM \rhd T_1 \times T_2 \mid_\chi C}$ Product Pattern Matching Judgement

$$\frac{}{X \rhd X_1 \times X_2 \mid_{\{X_1, X_2\}} \{X \doteq X_1 \times X_2\}} \qquad \frac{}{T_1 \times T_2 \rhd T_1 \times T_2 \mid_\emptyset \emptyset} \qquad \frac{}{\mathsf{Dyn} \rhd \mathsf{Dyn} \times\ \mathsf{Dyn} \mid_\emptyset \emptyset}$$

$\boxed{PM \rhd T_1 + T_2 \mid_\chi C}$ Sum Pattern Matching Judgement

$$\frac{}{X \rhd X_1 + X_2 \mid_{\{X_1, X_2\}} \{X \doteq X_1 + X_2\}} \qquad \frac{}{T_1 + T_2 \rhd T_1 + T_2 \mid_\emptyset \emptyset} \qquad \frac{}{\mathsf{Dyn} \rhd \mathsf{Dyn} + \mathsf{Dyn} \mid_\emptyset \emptyset}$$

Figure 5.7: Constraint Generation (Products and Sums)

Pattern matching requires the size of the tuple. In the type inference rule *Projection*, we can observe that the size of the tuple constructor is provided by the projection term (in π_i^n e, n denotes the size of the tuple). The same is true for pattern matching variants:

$$PM \rhd \langle l_i : T_i \ ^{i \in 1..n} \rangle \mid_\chi C$$

Instead of providing the size, the pattern matching relation requires the labels that form the variant type constructor. Variants with different labels are considered different types. These labels are provided by the labels in the alternatives in the case term. In the case of pattern

$\boxed{C \upsilon S}$ Constraint Solving

$$\frac{C \cup \{T_{11} \dot{\sim} T_{21}, T_{12} \dot{\sim} T_{22}\} \upsilon S}{C \cup \{T_{11} \times T_{12} \dot{\sim} T_{21} \times T_{22}\} \upsilon S} \qquad \frac{C \cup \{T_{11} \dot{\sim} T_{21}, T_{12} \dot{\sim} T_{22}\} \upsilon S}{C \cup \{T_{11} + T_{12} \dot{\sim} T_{21} + T_{22}\} \upsilon S}$$

$$\frac{\{X_1, X_2\} fresh \qquad X \notin Vars(T_1 \times T_2) \qquad C \cup \{X \doteq X_1 \times X_2, X_1 \dot{\sim} T_1, X_2 \dot{\sim} T_2\} \upsilon S}{C \cup \{X \dot{\sim} T_1 \times T_2\} \upsilon S} \qquad \frac{\{X_1, X_2\} fresh \qquad X \notin Vars(T_1 + T_2) \qquad C \cup \{X \doteq X_1 + X_2, X_1 \dot{\sim} T_1, X_2 \dot{\sim} T_2\} \upsilon S}{C \cup \{X \dot{\sim} T_1 + T_2\} \upsilon S}$$

$$\frac{C \cup \{T_{11} \doteq T_{21}, T_{12} \doteq T_{22}\} \upsilon S}{C \cup \{T_{11} \times T_{12} \doteq T_{21} \times T_{22}\} \upsilon S} \qquad \frac{C \cup \{T_{11} \doteq T_{21}, T_{12} \doteq T_{22}\} \upsilon S}{C \cup \{T_{11} + T_{12} \doteq T_{21} + T_{22}\} \upsilon S}$$

Figure 5.8: Constraint Solving (Products and Sums)

matching the recursive type, the pattern matching only requires the recursive variable (X).

$$PM \rhd \mu X \ . \ T \mid_{\chi} C$$

Pattern matching for the type constructors defined in this paper had to be derived, since there were no definitions available. The design was inspired by the Constraint Codomain Judgement and Constraint Domain Consistency Judgement in [14]. Deriving pattern matching judgements for a specific type constructor is somewhat straightforward. Like in the Constraint Codomain Judgement and Constraint Domain Consistency Judgement, we must take into account that the judgement expects a certain type constructor and can receive as input three different forms of types: a type variable, a type whose constructor is the expected and the dynamic type. Anything other than that leads the judgement to fail. When we try to pattern match a type variable, we must create a type whose structure is the expected type constructor and it's elements are (fresh) type variables. We then return that type, and an equality constraint between the newly created type and the original type variable. If we try to pattern match a type whose type constructor is the expected, then we can return that type and no constraints are needed. When we try to pattern match the dynamic type, we must return a type whose structure is that of the type constructor and it's elements are the dynamic type. The reason pattern matching judgements expect a type variable is because we are considering a type inference style approach. In a type system, a pattern matching relation only expects either the correct type constructor or the dynamic type.

$\boxed{\Gamma \vdash e : T \mid_{\chi} C}$ Constraint Generation

$$\frac{\text{for each } i \qquad \Gamma \vdash e_i : T'_i \mid_{\chi_i} C_i}{\Gamma \vdash (e_i\ ^{i\in 1..n}) \text{ as } (T_i\ ^{i\in 1..n}) : (T_i\ ^{i\in 1..n}) \mid_{\chi_1 \cup ... \cup \chi_n} C_1 \cup ... \cup C_n \cup \{T_1 \dot{\sim} T'_1, ..., T_n \dot{\sim} T'_n\}} \textsc{Tuple}$$

$$\frac{\Gamma \vdash e : PM \mid_{\chi} C \qquad PM \rhd (T_j\ ^{j\in 1..n}) \mid^{\chi'} C'}{\Gamma \vdash \pi_i^n\ e : T_i \mid_{\chi \cup \chi'} C \cup C'} \textsc{Projection}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : PM \mid_{\chi} C \qquad PM \rhd \langle l_i : T'_i\ ^{i\in 1..n}\rangle \mid_{\chi'} C' \\ \text{for each } i \qquad \Gamma, x_i : T'_i \vdash e_i : T_i \mid_{\chi_i} C_i \qquad T_1 \sqcap ... \sqcap T_n = T_J \mid_{\chi''} C''\end{array}}{\Gamma \vdash \text{case } e \text{ of } \langle l_i = x_i\rangle \Rightarrow e_i\ ^{i\in 1..n} : T_J \mid_{\chi \cup \chi' \cup \chi_1 \cup ... \cup \chi_n ... \cup \chi''} C \cup C' \cup C_1 \cup ... \cup C_n \cup C''} \textsc{Case}$$

$$\frac{\Gamma \vdash e_j : T'_j \mid_{\chi} C}{\Gamma \vdash \langle l_j = e_j\rangle \text{ as } \langle l_i : T_i\ ^{i\in 1..n}\rangle : \langle l_i : T_i\ ^{i\in 1..n}\rangle \mid_{\chi} C \cup \{T_j \dot{\sim} T'_j\}} \textsc{Tag}$$

$\boxed{PM \rhd (T_i\ ^{i\in 1..n}) \mid_{\chi} C}$ Tuple Pattern Matching Judgement

$$\frac{}{X \rhd (X_i\ ^{i\in 1..n}) \mid_{\{X_1,...,X_n\}} \{X \doteq (X_i\ ^{i\in 1..n})\}} \qquad \frac{}{(T_i\ ^{i\in 1..n}) \rhd (T_i\ ^{i\in 1..n}) \mid_{\emptyset} \emptyset}$$

$$\frac{}{\mathsf{Dyn} \rhd (\mathsf{Dyn}\ ^{i\in 1..n}) \mid_{\emptyset} \emptyset}$$

$\boxed{PM \rhd \langle l_i : T_i\ ^{i\in 1..n}\rangle \mid_{\chi} C}$ Variant Pattern Matching Judgement

$$\frac{}{X \rhd \langle l_i : X_i\ ^{i\in 1..n}\rangle \mid_{\{X_1,...,X_n\}} \{X \doteq \langle l_i : X_i\ ^{i\in 1..n}\rangle\}} \qquad \frac{}{\langle l_i : T_i\ ^{i\in 1..n}\rangle \rhd \langle l_i : T_i\ ^{i\in 1..n}\rangle \mid_{\emptyset} \emptyset}$$

$$\frac{}{\mathsf{Dyn} \rhd \langle l_i : \ \mathsf{Dyn}\ ^{i\in 1..n}\rangle \mid_{\emptyset} \emptyset}$$

Figure 5.9: Constraint Generation (Tuples and Variants)

$\boxed{C \upsilon S}$ Constraint Solving

$$\frac{C \cup \{T_{11} \dot{\sim} T_{21}, ..., T_{1n} \dot{\sim} T_{2n}\} \upsilon S}{C \cup \{(T_{1i}{}^{i\in 1..n}) \dot{\sim} (T_{2i}{}^{i\in 1..n})\} \upsilon S} \qquad \frac{C \cup \{T_{11} \dot{\sim} T_{21}, ..., T_{1n} \dot{\sim} T_{2n}\} \upsilon S}{C \cup \{\langle l_{1i} : T_{1i}{}^{i\in 1..n}\rangle \dot{\sim} \langle l_{2i} : T_{2i}{}^{i\in 1..n}\rangle\} \upsilon S}$$

$$\frac{\{X_1, ..., X_n\} fresh \qquad X \notin Vars((T_i{}^{i\in 1..n}))}{C \cup \{X \doteq (X_i{}^{i\in 1..n}), X_1 \dot{\sim} T_1, ..., X_n \dot{\sim} T_n\} \upsilon S}{C \cup \{X \dot{\sim} (T_i{}^{i\in 1..n})\} \upsilon S}$$

$$\frac{\{X_1, ..., X_n\} fresh \qquad X \notin Vars(\langle l_i : T_i{}^{i\in 1..n}\rangle)}{C \cup \{X \doteq \langle l_i : X_i{}^{i\in 1..n}\rangle, X_1 \dot{\sim} T_1, ..., X_n \dot{\sim} T_n\} \upsilon S}{C \cup \{X \dot{\sim} \langle l_i : T_i{}^{i\in 1..n}\rangle\} \upsilon S} \qquad \frac{C \cup \{T_{11} \doteq T_{21}, ..., T_{1n} \doteq T_{2n}\} \upsilon S}{C \cup \{(T_{1i}{}^{i\in 1..n}) \doteq (T_{2i}{}^{i\in 1..n})\} \upsilon S}$$

$$\frac{C \cup \{T_{11} \doteq T_{21}, ..., T_{1n} \doteq T_{2n}\} \upsilon S}{C \cup \{\langle l_{1i} : T_{1i}{}^{i\in 1..n}\rangle \doteq \langle l_{2i} : T_{2i}{}^{i\in 1..n}\rangle\} \upsilon S}$$

Figure 5.10: Constraint Solving (Tuples and Variants)

$\boxed{\Gamma \vdash e : T \mid_\chi C}$ Constraint Generation

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash e : T' \mid_\chi C}{\Gamma \vdash \mathsf{fold}\ [U]\ e : U \mid_\chi C \cup \{[X \mapsto U]T \dot{\sim} T'\}}\ \textsc{Fold}$$

$$\frac{U = \mu X\ .\ T \qquad \Gamma \vdash e : PM \mid_\chi C \qquad PM \rhd \mu X'\ .\ T' \mid_{\chi'} C'}{\Gamma \vdash \mathsf{unfold}\ [U]\ e :\ [X \mapsto U]T \mid_{\chi \cup \chi'} C \cup C' \cup \{U \dot{\sim} \mu X'\ .\ T'\}}\ \textsc{Unfold}$$

$\boxed{PM \rhd \mu X\ .\ T \mid_\chi C}$ Recursive Type Pattern Matching Judgement

$$\frac{}{X' \rhd \mu X\ .\ X_1 \mid_{\{X_1\}} \{X' \doteq \mu X\ .\ X_1\}} \qquad \frac{}{\mu X\ .\ T \rhd \mu X\ .\ T \mid_\emptyset \emptyset} \qquad \frac{}{\mathsf{Dyn} \rhd \mu X\ .\ \mathsf{Dyn} \mid_\emptyset \emptyset}$$

Figure 5.11: Constraint Generation (Recursive types)

$\boxed{C \upsilon S}$ Constraint Solving

$$\frac{C \cup \{T_1 \dot{\sim} T_2\} \upsilon S}{C \cup \{\mu X \,.\, T_1 \dot{\sim} \mu X \,.\, T_2\} \upsilon S} \qquad \frac{\{X_1\} fresh \qquad X \notin Vars(\mu X' \,.\, T) \qquad C \cup \{X \doteq \mu X' \,.\, X_1, X_1 \dot{\sim} T\} \upsilon S}{C \cup \{X \dot{\sim} \mu X' \,.\, T\} \upsilon S}$$

$$\frac{C \cup \{T_1 \doteq T_2\} \upsilon S}{C \cup \{\mu X \,.\, T_1 \doteq \mu X \,.\, T_2\} \upsilon S}$$

Figure 5.12: Constraint Solving (Recursive types)

Chapter 6

Conclusion

Gradual typing allows the programmer to choose which typing discipline is used in a specific part of the program: static or dynamic typing, thus allowing a program to retain the advantages of both disciplines. However, gradual typing requires specific type system rules, cast insertion rules and evaluation rules, along with other mechanisms that support gradual typing and the existence of the dynamic type, which are not easily derived. In this dissertation, we focus on these new systems and rules, providing an overview and a detailed explanation of the different components that allow gradual typing, and we show how these components interact with each other. To further demonstrate the interconnection of these components, we also implement an interpreter in Haskell for a gradual language with type inference. Finally we extend the previous work on gradual typing with gradual algebraic data types, which are regular algebraic data types with the possibility for dynamic elements.

Future work includes parametric polymorphism, overloading (with type classes) and other type systems, such as dependent types and how can these systems be changed to allow gradual typing.

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM transactions on programming languages and systems (TOPLAS)*, 13(2):237–268, 1991.

[2] Pedro Ângelo, Mário Florido, and Pedro Vasconcelos. Gradual compound data types. Technical report, LIACC-UP, 2017.

[3] Arthur I Baars and S Doaitse Swierstra. Typing dynamic typing. In *ACM SIGPLAN Notices*, volume 37, pages 157–166. ACM, 2002.

[4] Henk Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(02):181–215, 1997.

[5] HP Barendregt. The lambda calculus: Its syntax and semantics, volume 103 of studies in logic and the foundations of computer science, 1981.

[6] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[7] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(02):56–68, 1940.

[8] Alonzo Church. *The calculi of lambda-conversion*. Number 6. Princeton University Press, 1941.

[9] Matteo Cimini and Jeremy G Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. *ACM SIGPLAN Notices*, 51(1):443–455, 2016.

[10] Matteo Cimini and Jeremy G Siek. Automatically generating the dynamic semantics of gradually typed languages. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 789–803. ACM, 2017.

[11] Haskell B Curry, Robert Feys, and William Craig. Combinatory logic, volume i. 1959.

[12] Luis Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, 1984.

[13] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[14] Ronald Garcia and Matteo Cimini. Principal type schemes for gradual programs. In *ACM SIGPLAN Notices*, volume 50, pages 303–315. ACM, 2015.

[15] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. Sage: Practical hybrid checking for expressive types and specifications. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 93–104, 2006.

[16] Fritz Henglein. Dynamic typing. In *ESOP'92*, pages 233–253. Springer, 1992.

[17] Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.

[18] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[19] Daan Leijen. Parsec, a fast combinator parser, 2001.

[20] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[21] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and practice of object systems*, 5(LAMP-ARTICLE-1999-001):35, 1999.

[22] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *Exploring new frontiers of theoretical informatics*, pages 437–450. Springer, 2004.

[23] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[24] J Alan Robinson. Computational logic: The unification computation. *Machine intelligence*, 6(63-72):10–1, 1971.

[25] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[26] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[27] Jeremy G Siek and Manish Vachharajani. Gradual typing with unification-based inference. In *Proceedings of the 2008 symposium on Dynamic languages*, page 7. ACM, 2008.

[28] Jeremy G Siek, Michael M Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[29] Satish Thatte. Type inference with partial types. In *International Colloquium on Automata, Languages, and Programming*, pages 615–629. Springer, 1988.

[30] Satish Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 367–381. ACM, 1989.

[31] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 964–974. ACM, 2006.

[32] Laurence Tratt. Dynamically typed languages. *Advances in Computers*, 77:149–184, 2009.

[33] Philip Wadler. A prettier printer. *The Fun of Programming, Cornerstones of Computing*, pages 223–243, 2003.

[34] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10(2):115–121, 1987.

Appendix A

Type Systems in Prolog

The implementation described in this appendix is fully available in the GitHub repository https://github.com/pedroangelo/gradualizer.

A.1 Simply Typed Lambda Calculus (STLC)

```
-- STLC Signatures
abs :: Term -> Term -> Term
app :: Term -> Term -> Term
arrow :: Type -> Type -> Type


%% Simply Typed Lambda Calculus type system
type(Context, var(X), T) :- member(X:T, Context).
type(Context, abs(X, E), arrow(T1, T2)) :- type([X:T1 | Context], E, T2).
type(Context, app(E1, E2), B) :- type(Context, E1, arrow(A, B)),
                                 type(Context, E2, A).
```

A.2 STLC with Addition

```
-- STLC with Addition Signatures
zero :: Term
succ :: Term -> Term
add :: Term -> Term -> Term
int :: Type


%% Extension to Simply Typed Lambda Calculus type system
%% Addition and Integers
type(Context, zero, int).
type(Context, succ(E), int) :- type(Context, E, int).
type(Context, add(E1, E2), int) :- type(Context, E1, int),
```

```
                                    type(Context, E2, int).
```

A.3 STLC with Lists

```
-- STLC with Lists Signatures
emptyList :: Type -> Term
isnil :: Type -> Term -> Term
cons :: Type -> Term -> Term -> Term
head :: Type -> Term -> Term
tail :: Type -> Term -> Term
list :: Type -> Type
true :: Term
false :: Term
bool :: Type


%% Extension to Simply Typed Lambda Calculus type system
%% Lists
type(Context, emptyList(T), list(T)).
type(Context, isnil(T, E), bool) :- type(Context, E, list(T)).
type(Context, cons(T, E1, E2), list(T)) :- type(Context, E1, T),
                                           type(Context, E2, list(T)).
type(Context, head(T, E), T) :- type(Context, E, list(T)).
type(Context, tail(T, E), list(T)) :- type(Context, E, list(T)).
type(Context, true, bool).
type(Context, false, bool).
```

Appendix B

Gradualizer Implementation

The implementation described in this appendix is fully available in the GitHub repository https://github.com/pedroangelo/gradualizer.

```
module Gradualizer (
   generateGradual,
   generateCompiler,
   gradualize,
   compilerToCC
) where

-- Type Systems
import InitialTypeSystem as ITS
import GradualTypeSystem as GTS
import CastCalculus as CC

-- Parsers
import PrologParser
import SignatureParser

-- Gradulizer Steps
import Classify
import PatternMatch
import Flows
import LoneInputs
import Consistency
import PatternMatchRules
import CastInsertion

-- Imports
import Data.Maybe
```

```
-- read type system from prolog file,
-- parse contents and convert to TypeSystem,
-- then gradualize
generateGradual :: FilePath -> IO GTS.TypeSystem
generateGradual file = do
   -- parse prolog to Program format
   prolog <- parseProlog file
   -- parse signatures to Signatures format
   signatures <- parseSignatures file
   -- get type annotated terms
   let typeAnnotatedTerms = deriveTypeAnnotations signatures
   -- convert Program format to Initial Type System using annotated terms
       information
   let ts = ITS.toTypeSystem prolog typeAnnotatedTerms
   -- gradualize typesystem
   return $ gradualize signatures ts

-- gradualize type system by applying 6 steps:
-- - Step 1: Classify type variables with input or output modes
-- - Step 2: Classify type variables with producer or consumer positions
-- - Step 3: Apply pattern matching to constructed outputs
-- - Step 4: Flow and final type discovery
-- - Step 5: Restrict lone inputs to be static
-- - Step 6: Replace flow with consistency
-- - Step PM: Add pattern matching rules
-- - Step Final: Remove mode and position from type variables and flow relation
    from typing relation
gradualize :: Signatures -> ITS.TypeSystem -> GTS.TypeSystem
gradualize signatures =
   -- step Final
   GTS.convertTypeSystem .
   -- step PM
   addPatternMatchRules signatures .
   -- step 6
   removeFlowsInsertConsistency .
   -- step 5
   loneInputsStatic .
   -- step 4
   insertFlowsFinalType .
   -- step 3
   applyPatternMatching .
   -- step 1 and 2
   classifyTypeVariables
```

```
-- read type system from prolog file,
-- parse contents and convert to TypeSystem,
-- then generate compiler to cast calculus
generateCompiler :: FilePath -> IO CC.TypeSystem
generateCompiler file = do
   -- parse prolog to Program format
   prolog <- parseProlog file
   -- parse signatures to Signatures format
   signatures <- parseSignatures file
   -- get type annotated terms
   let typeAnnotatedTerms = deriveTypeAnnotations signatures
   -- convert Program format to Initial Type System using annotated terms
       information
   let ts = ITS.toTypeSystem prolog typeAnnotatedTerms
   -- generate compiler to cast calculus
   return $ compilerToCC signatures ts

-- generate a compiler to cast calculus by applying 6 steps:
-- - Step 1: Classify type variables with input or output modes
-- - Step 2: Classify type variables with producer or consumer positions
-- - Step 3: Apply pattern matching to constructed outputs
-- - Step 4: Flow and final type discovery
-- - Step 5: Restrict lone inputs to be static
-- - Step 7: Generate casts as directed by the flow premises
-- - Step 6: Replace flow with consistency
-- - Step Final: Remove mode and position from type variables and flow relation
    from typing relation
compilerToCC :: Signatures -> ITS.TypeSystem -> CC.TypeSystem
compilerToCC signatures =
   -- step Final
   CC.convertTypeSystem .
   -- step 6
   removeFlowsInsertConsistency .
   -- step 7
   generateCasts .
   -- step 5
   loneInputsStatic .
   -- step 4
   insertFlowsFinalType .
   -- step 3
   applyPatternMatching .
   -- step 1 and 2
   classifyTypeVariables
```

Appendix C

Parser

The implementation described in this appendix is fully available in the GitHub repository https://github.com/pedroangelo/interpreter.

```
module Parser (
   expressionParser,
   typeParser
) where

-- Syntax & Types
import Syntax
import Types
import Examples

-- Parsec
import Text.Parsec hiding (label, labels)
import Text.Parsec.String (Parser)
import Text.ParserCombinators.Parsec.Expr
import Text.ParserCombinators.Parsec.Language (emptyDef)
import Text.ParserCombinators.Parsec.Combinator
import qualified Text.Parsec.Token as Token

-- language definition for lexer
languageDefinition :: Token.LanguageDef ()
languageDefinition = emptyDef {
   Token.commentStart      = "{-",
   Token.commentEnd        = "-}",
   Token.commentLine       = "--",
   Token.nestedComments = False,
   Token.identStart        = lower,
   Token.identLetter       = alphaNum <|> oneOf "_'",
   Token.opStart           = oneOf ":!#$%&*+./<=>?@\\^|-~",
   Token.opLetter          = oneOf ":!#$%&*+./<=>?@\\^|-~",
```

```
    Token.reservedNames     = ["true", "false", "let", "in", "fix", "letrec",
                               "if", "then", "else", "unit", "as", "case", "of",
                               "nil", "cons", "isnil", "head", "tail", "fold",
                               "unfold", "Int", "Bool", "Unit", "Dyn", "forall", "rec"],
    Token.reservedOpNames   = ["\\", ".",":","::","=","+","-","*","-","/","==",
                               "/=","<",">","<=",">=","->","{","}","[","]"],
    Token.caseSensitive     = True }

-- Expression Parser
expressionParser = do
   whiteSpace
   e <- expression
   whiteSpace
   eof
   return e

-- expressions
expression :: Parser Expression
expression =
   try abstractionExpression <|>
   try ascriptionExpression <|>
   annotationExpression <|>
   letExpression <|>
   fixExpression <|>
   letrecExpression <|>
   ifExpression <|>
   try projectionTupleExpression <|>
   try projectionRecordExpression <|>
   caseExpression <|>
   nilExpression <|>
   try emptyListExpression <|>
   consExpression <|>
   try consOperatorExpression <|>
   listExpression <|>
   isNilExpression <|>
   headExpression <|>
   tailExpression <|>
   foldExpression <|>
   unfoldExpression <|>
   operatorExpression <?> "expression"
```

```
-- force parentheses in some expressions
parensExpression :: Parser Expression
parensExpression =
   variableExpression <|>
   intExpression <|>
   boolExpression <|>
   unitExpression <|>
   try tupleExpression <|>
   try recordExpression <|>
   tagExpression <|>
   parens expression <?> "delimited expression"

-- operators
operatorExpression :: Parser Expression
operatorExpression = buildExpressionParser operators applicationExpression

-- application
applicationExpression :: Parser Expression
applicationExpression = do
   { es <- many1 parensExpression
   ; return $ foldl1 Application es } <?> "application"

-- variable
variableExpression :: Parser Expression
variableExpression = do
   { x <- identifier
   ; return $ Variable x } <?> "variable"

-- abstraction
abstractionExpression :: Parser Expression
abstractionExpression = do
   { reservedOp "\\"
   ; v <- identifier
   ; reservedOp "."
   ; e <- expression
   ; return $ Abstraction v e } <?> "abstraction"

-- Type Parser
typeParser = typ

typ :: Parser Type
typ =
   arrowOperator
```

```
parensType :: Parser Type
parensType =
   varType <|>
   intType <|>
   boolType <|>
   unitType <|>
   dynType <|>
   muType <|>
   try tupleType <|>
   recordType <|>
   variantType <|>
   listType <|>
   parens typ

-- Type variable: Var
varType :: Parser Type
varType = do
   { var <- identifier
   ; return $ VarType var } <?> "type variable"

-- Arrow type: Type -> Type
arrowOperator :: Parser Type
arrowOperator = buildExpressionParser
   [[binary "->" ArrowType AssocRight]] parensType
```

Appendix D

Interpreter

The implementation described in this appendix is fully available in the GitHub repository https://github.com/pedroangelo/interpreter.

```
module Interpreter (
    interpret,
    runCode,
    runFile
) where

-- Syntax & Types
import Syntax
import Types
import Examples

-- Type Inference
import TypeInference

-- Cast Insertion
import CastInsertion

-- Evaluation
import Evaluation

-- Pretty Printing
import PrettyPrinter

-- Parser
import Parser

-- Imports
import Data.Either
import Text.Parsec
```

```
-- run interpreter for given expression
interpret :: Expression -> IO ()
interpret expr = interpretCode False expr

-- run interpreter for given expression
interpretCode :: Bool -> Expression -> IO ()
interpretCode parameter expr = do
   let prettyE = if parameter then (\x -> show $ prettyExpression x) else show
   let prettyT = if parameter then (\x -> show $ prettyType x ) else show
   -- get type of expression
   let ti = inferType expr
   -- if expression is ill typed
   if isLeft ti then do
      -- print error
      let (Left err) = ti
      putStrLn err
      return ()
   -- if expression is well typed
   else do
      -- print expression
      putStrLn $ "Expression: " ++ prettyE expr
      -- print type
      let (Right typ) = ti
      putStrLn $ "\nExpression type: " ++ prettyT typ
      -- insert casts
      let (Right typedExpr) = insertTypeInformation expr
      let casts = removeTypeInformation $ insertCasts typedExpr
      putStrLn $ "\nCast insertion: " ++ prettyE casts
      -- run evaluation
      let result = evaluate casts
      putStrLn $ "\nEvaluation result: " ++ prettyE result

runCode :: String -> IO ()
runCode string = do
   -- parse string
   let parsed = parse expressionParser "" string
   -- if parsing failed
   if isLeft parsed then do
      -- print error
      let (Left parseError) = parsed
      putStrLn $ show parseError
      return ()
   else do
      -- get expression
      let (Right expr) = parsed
      -- run interpreter
      interpretCode True expr
```

```
runFile :: FilePath -> IO ()
runFile filePath = do
   -- get file contents
   fileContents <- readFile filePath
   -- parse contents
   let parsed = parse expressionParser "" fileContents
   -- if parsing failed
   if isLeft parsed then do
      -- print error
      let (Left parseError) = parsed
      putStrLn $ show parseError
      return ()
   else do
      -- get expression
      let (Right expr) = parsed
      -- run interpreter
      interpretCode True expr
```